University of Alberta

APPROXIMATION ALGORITHMS FOR FREQUENCY RELATED QUERY PROCESSING ON STREAMING DATA

by

©

Fan Deng

A thesis submitted to the Faculty of Graduate Studies and Research in partial
fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Department of Computing Science

Edmonton, Alberta

Fall 2007

Library and
Archives Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-32950-4
Our file Notre référence
ISBN: 978-0-494-32950-4

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

To my wife, Yina Liu, and to my parents, Rugang Deng and Anruo Li.

Abstract

With the emergence of more and more data stream applications, such as Internet traffic measurement, sensor network data monitoring, Web click and crawling stream analysis, financial data alert and so on, researchers realize that many data processing models and algorithms well-suited for traditional database applications are not applicable in those new streaming scenarios. To address the problem, novel data stream processing systems and algorithms have been proposed within database, theory and computer networking communities. Some of the work has led to commercial systems or algorithms which have been applied in industry. Streaming algorithms also have found their way into non-streaming environments where massive data processing is needed.

This thesis focuses on the algorithmic aspect of data stream processing, more specifically, approximation algorithms for answering frequency related queries on streaming data. Example queries are "find the number of similar record pairs in a very large relational table", "identify URLs that appear for the first time in the crawling stream of a search engine" and "give a list of IDs which appear more frequently in a web click stream". Efficiently answering these queries with bounds on time and space costs is both important and challenging, because fast response is either required or desirable in many scenarios, and the available computing and fast storage resources are often very limited compared with the massive streaming data volume. Thus, approximation algorithms trading accuracy for computing or storage resources is a valuable option in these cases.

In this thesis, two types of data reduction techniques, namely sketching

and sampling, are studied. Both techniques are suitable not only for answering frequency related queries over streaming data, but also have a wide range of other application areas. This thesis focuses on exploring these powerful techniques to answer frequency related queries under data stream scenarios. More precisely, based on well-known sketching and sampling techniques, this thesis proposes new data structures and one-pass approximation algorithms to answer membership queries, frequency queries, join/self-join size estimations, similarity join/self-join size estimations and result set size estimations for similarity searches. Both theoretical and experimental analysis show that our new techniques improve the state-of-the-art.

Acknowledgements

First, I have to thank my supervisor, Dr. Davood Raifei. He has spent so much time on my research, not only giving high level research advices, but also discussed detailed technical issues with me. From those meetings, which often takes several hours each, I've learned a lot about how to do research, from general methodology to specific knowledge. Especially in the beginning years of my Ph.D. program, he was so patient of my naiveness and unconfidence of research, which was crucial for me to grow up. More importantly, I really appreciate the freedom he gives me, which makes me can experience the joy of research during the thinking and learning process. I believe these experiences will affect my whole life.

I would like to thank Dr. Mario A. Nascimento, Dr. M.H. MacGregor, Dr. Marek Reformat for their constructive comments and precious time on my research. Also, thanks Dr. Joerg Sander, Dr. Mario A. Nascimento, Dr. Li-yan Yuan for their cooperation and tolerance in my teaching assistant work so that I can efficiently schedule my time between research and teaching. And I feel lucky that I have work with several great partners in CMPUT391: Varun Grover, Chihoon Lee, Hongyu Zhang, Yunping Wang.

Also, I am really glad to have known so many great friends during my Ph.D. program. Reza Sherkat and I had numerous discussions over the past five years on different topics including our research, lives and future. Thanks Qiang Ye and Tao Wang, my officemates, for giving me lots of advices and help in getting used to Edmonton and the department when I just came to Canada. Thanks to my great lab mates, Chonghai Wang, Pirooz Chubak, Amit Satsangi, Pouria Pirzadeh and Vahid Jazayeri, made our DB lab a very pleasing working space. Although Alexandru Coman, Jianjun Zhou, Catalina Maria-Luiza Antonie, Gabriela Moise, Stanley Oliveira, Mohammad El-Hajj, Samer Nassar, either came to the lab less often or have already left Edmonton, I really enjoyed the communication with them.

Last, thanks the Department of Computing Science and the University of Alberta for providing me an excellent research environment.

Table of Contents

1 Introduction 1
1.1 Data Stream Applications 1
1.1.1 Network Traffic Monitoring and Measurement 2
1.1.2 Real-time Financial Data Analysis 4
1.1.3 Streams from the Web 4
1.2 Traditional Data Processing V.S. Data Stream Applications . 5
1.3 Data Stream Model 7
1.3.1 Tuple-based and Time-based Stream 7
1.3.2 One-time Query and Continuous Query 7
1.3.3 Query Windows 7
1.4 Data Stream Research Sketch 8
1.4.1 Data Stream Systems 8
1.4.2 Data Stream Algorithms 10
1.5 Thesis Overview 10
1.5.1 Thesis Scope 10
1.5.2 Contributions 11

2 Sketching and Sampling 13
2.1 Sketching 13
2.1.1 Bloom Filters and their extensions 14
2.1.2 FM Sketches and their extensions 17
2.1.3 AMS Sketches and extensions 19
2.2 Sampling 20
2.2.1 Random Sampling 20
2.2.2 Distinct Sampling 21
2.2.3 Count sample and sticky sample 21

3 Approximate Membership Query Processing 23
3.1 Approximate Membership Query 24
3.1.1 Motivating Examples 25
3.2 Preliminaries 26
3.2.1 Problem Statement 27
3.2.2 The Buffering Method 27
3.3 Stable Bloom Filters 28
3.3.1 The Challenge to Bloom Filters 28
3.3.2 Our Approach 28
3.3.3 The Stable Property 30
3.3.4 The Stable Point 34
3.3.5 False Positive Rates 39
3.3.6 False Negative Rates 40

3.4 From Theory to Practice 42
3.4.1 Setting Parameters 42
3.4.2 Time Complexity 49
3.5 Experiments 50
3.5.1 Data Sets 50
3.5.2 Implementation Issues 52
3.5.3 Theory Verification 54
3.5.4 Error Rates Comparison 54
3.5.5 Time Comparison 59
3.5.6 Methods Comparison Summary 61
3.6 Related Work 62
3.7 Summary and Possible Extension 63

4 Approximate Frequency Query Processing 65
4.1 Frequency Queries and Multipurpose Sketching. 66
4.2 Unbiased Estimates for Multiplicity Queries using Count-min Sketches 68
4.2.1 Basic Idea 68
4.2.2 Our Estimation Algorithm 69
4.2.3 Analyses of Our Algorithm 70
4.2.4 Experiments for Multiplicity Queries 74
4.2.5 Summary of Comparisons 79
4.3 Unbiased Self-join Size Estimates from Count-min Sketches . . 80
4.3.1 Our Estimation Algorithm 81
4.3.2 Analyses of Our Algorithm 81
4.3.3 Experiments for Self-join Size Estimations 85
4.4 Related Work 86
4.5 Summary and Potential Extension 87

5 Efficient Result Set Size Estimation for Similarity Queries on Large Data Sets 91
5.1 Similarity Join/Self-join and Selectivity Estimation 91
5.2 Similarity Self-join Size Estimation 95
5.2.1 Problem Statement 96
5.2.2 A Straightforward Solution 96
5.2.3 Random sampling 97
5.2.4 Our SelfJoinPairCount Algorithm (Offline Scenario) . . 97
5.2.5 Our SelfJoinPairCount Algorithm (Online Scenario) . . 104
5.3 Result Set Size Estimation for Similarity Join and Search . . . 109
5.3.1 Similarity Join Size Estimation 109
5.3.2 Our JoinPairCount Algorithm 110
5.3.3 Result Set Size Estimation for Similarity Search 112
5.4 Extensions and Applications 114
5.5 Experiments 116
5.5.1 Experimental Setup 116
5.5.2 Offline SelfJoinPairCount 116
5.5.3 Online SelfJoinPairCount 118
5.6 Related Work 120
5.7 Summary and Possible Extension 122

6 Conclusions and Future Work 123

Bibliography 125

List of Tables

3.1 The Symbol List 30

5.1 SelfJoinPairCount relative errors (estimate-truth)/truth. . . . 117
5.2 SelfJoinPairCount relative errors with different data set sizes. 117
5.3 SelfJoinPairCount relative errors with different sampling factors. 117
5.4 SelfJoinPairCount relative errors (estimate-truth)/truth with different dimensionality. 118
5.5 Online SelfJoinPairCount relative errors (estimate-truth)/truth. 119
5.6 Online SelfJoinPairCount relative errors with different data set sizes. 119
5.7 Online SelfJoinPairCount and random sample relative errors. . 120

List of Figures

3.1 FN rates vs. K 44
3.2 FN rates difference between $Max = 3$ and $Max = 1$ ($Max3 - Max1$) vs. gaps. K is set to the optimal value respectively under different settings. 48
3.3 Fraction of zeros changed with time on the whole real data set (Max=1, K=2, P=4, FPS=10%), space unit=64bits 55
3.4 Error rates comparison between SBF, FPBuffering, Buffering and BF 56
3.5 FN rate differences between FPBuffering and SBF varying allowable FP rate(695M elements) 58
3.6 FN rate comparisons between SBF and FPBuffering on synthetic data sets (FP rates $\leq 10\%$, in the SBF method $Max = 1$, $K = 2$ and $P = 4$). 60

4.1 Average absolute errors vs. data set skew, comparing Fast-AGMS, CM, MI and our CMM; queries are all elements in the domain. The sketch width and depth are 64 and 3 respectively. 75
4.2 Average absolute errors vs. data set skew, for top-100 frequent element queries using Fast-AGMS, CM, CMM and MI. The sketch width and depth are 256 and 5 respectively. 76
4.3 Average absolute errors vs. sketch width, for top-100 frequent element queries using Fast-AGMS, CM, CMM and MI on the 1M URL data set. The sketch depth is 5. 77
4.4 Average absolute errors vs. sketch depth, for top-100 frequent element queries using Fast-AGMS, CM, CMM and MI on the 1M URL data set. The sketch width is 256. 78
4.5 Self-join size estimation errors vs. data set skew, comparing Fast-AGMS, CM and CMM. The sketch width and depth are 16 and 5 respectively. (The 2nd sub-figure zooms in the 1st one). The stream sizes are all 1 million. 88
4.6 Self-join size estimation error vs. sketch width, comparing Fast-AGMS, CM and CMM on the 1M URL data set. The sketch depth is 3. The 2nd sub-figure zooms in the 1st one. 89
4.7 Self-join size estimation error vs. sketch depth, comparing Fast-AGMS, CM and CMM on the 1M URL data set. The sketch width is 16. 90

Chapter 1

Introduction

In this chapter, we present a few data stream processing applications, review some background and related works, and discuss the outline of our research and its contributions.

1.1 Data Stream Applications

A number of applications sharing certain common properties such as the one pass requirement and fast data arrival rates can be classified into data stream applications. These applications include IP traffic monitoring and measurement, real-time stock analysis, Web click stream processing, telephone call record analysis, sensor network monitoring, real-time server workload measurement, XML streams filtering, online auction bid stream monitoring and so on. While these applications have some properties in common, they also have some distinct characteristics in terms of data arrival rates, processing speed requirements, query types, accuracy requirements and so on. Next, we discuss some of these applications in detail.

1.1.1 Network Traffic Monitoring and Measurement

The Internet is becoming increasingly important and ubiquitous in today's business, science, education and entertainment, almost touching every aspect of our daily lives. It is also growing and changing: new applications emerge everyday, and technologies and users push the Internet to new territories.

To better understand the dynamics of a network and to react to changes, there is an increasing demand for network traffic monitoring and measurement. Following are some examples [74, 57, 26].

- Network engineering. Internet Service Providers (ISP) can monitor the network traffic and alleviate network congestions based on the gathered information. For example, network operators can change the routing configuration and reroute part of traffic away from congested areas.

- Network usage-based billing. When an ISP charge customers, network traffic to or from customers can directly or indirectly affect the billing policy.

- Network intrusion detection. Network attacks often lead to changes in traffic patterns. For example, there will be a large number of small flows from different IP addresses forwarded to a host under a DDos attack.

- Network topology and capacity planning. After monitoring or measuring the network for weeks or months, ISPs may consider upgrading their devices to eliminate the bottlenecks or change their network topology to improve the performance.

Challenges for Internet traffic monitoring and measurement. First, with the improvement of hardware devices, the volume and speed of Internet traffic keep growing. Routers that can forward gigabits of IP packets per second are common nowadays. A large ISP may collect data from tens of thousands of network interfaces [57]. A single high speed network interface could generate hundreds of gigabytes of unsampled flow statistics per day if fully utilized.

It has been reported that ordinary memory (DRAM) is sometimes too slow for per packet processing, while fast memory (SRAM) is quite expensive [63, 12]. For example, to process a 40-byte packet at a 40Gbps speed , a router has only 8 nanoseconds. DRAM access time (40 nanosecond) is obviously too slow. On-chip(1-2 nanoseconds access time) and off-chip SRAM (2-5 nanoseconds access time) are faster. But on-chip SRAM is limited (say 4 megabytes) and off-chip SRAM is very expensive [63].

Second, the current mechanisms provided for Internet measurements are very limited. The main task of a network is to deliver messages. In the original design of the Internet, only highly aggregated measurements were considered, and today only aggregate loss and utilization statistics are ubiquitously reported by router interfaces [57]. Obtaining detailed measurement information without incurring too much overhead is not easy.

Third, the resources that can be allocated for Internet measurements are also limited. Due to the huge data volume and high forwarding rates, very few CPU cycles and fast memory can be used for traffic measurement at routers. Usually, a portion of IP packet headers are collected and sent to a remote machine to be further analyzed. The required network bandwidth to transmit data from a collecting point to an analyzing machine is also high. Even at the remote data analyzing point, processing the transmitted data stream is not easy: it may need large amount of memory, huge secondary storage and powerful CPU.

Current Internet measurement techniques. Due to the speed constraints in data gathering and processing, different aggregation granularities including packet level, flow level and SNMP level aggregations may be applied [100]. At the finest level, IP packet headers such as source/destination IP addresses, source/destination port numbers, protocol types and so on are extracted. A higher level aggregation may be applied to flows, where a flow is a sequence of packets sharing some common properties such as IP addresses, port numbers or protocol types during a period of time [25, 106]. Flow measurements include flow identifiers, start time, end time, number of packets and bytes and so on. At the highest level, SNMP collects traffic volume statistics over links every few minutes. For packet and flow level aggregation, sampling is widely used in network measurement and accounting applications. Although sampling is already accepted as a standard solution for flow measurement [106, 105], it does not mean that it cannot be improved. Currently, active research is conducted on this topic in the network community [61, 62].

1.1.2 Real-time Financial Data Analysis

Another data stream application is monitoring real-time financial data such as stock ticks and foreign exchange rates. There are about 50,000 securities trading in the United States. Although the data generation rate is not as fast as that of IP packets, every second up to 100,000 quotes and trades (ticks) are generated based on a 2002 figure [121].

Real-time financial data analysis is becoming important. With the increase of trading speed, in addition to long-term trading, intra-day trading strategy is also widely used to make small profits in a short period, especially in electronic trading markets such as the NASDAQ. Uncovering momentary opportunities are crucial in this trading policy [90]. Therefore, real-time financial data monitoring tools are becoming more and more important for short term traders. Some sample queries on real-time financial data are posted at [113]. One example is "Find all stocks between $20 and $600 where the spread between the high tick and the low tick over the past 30 minutes is greater than 3% of the last price and in the last 5 minutes the average volume has surged by more than 300%."

1.1.3 Streams from the Web

URL and Document Streams of a Web Crawler. Search engines regularly crawl the Web to enlarge their collections of Web pages. During the crawling, some processing may be performed over the streams of crawled URLs and Web documents. For example, given the URL of a page which is extracted from the content of a crawled page, a search engine must probe its archive to find out if the URL is in the engine collection and if the fetching of the URL can be avoided [24, 78]. If the URL is not found in the archive, the crawler has to download this page from the Web.

Web Click Streams. Many individuals or companies who put their information on the Web care about the popularity of their web pages. Sometimes they are even willing to pay for the clicks to gain popularities [96]. In these cases, monitoring the web page clicks is quite important. Such statistics can be generated from the stream of source addresses of the users who click the web page address.

1.2 Traditional Data Processing V.S. Data Stream Applications

In the past few decades, many data processing techniques have been developed in the database community, but most of them are not suitable in a streaming environment for the following reasons.

First, traditional database applications store data on disks before queries arrive. The users assume that all data is available. The problem is how to find the information and answer the queries efficiently, and how to maintain the data. In contrast, most data stream applications do not require storing all the data, and sometimes it is hard to record the whole data stream because of the data arrival rate. For example, in Internet measurement applications, monitoring devices may not be able to keep up with the high speed IP packet forwarding rate using ordinary RAM. It is hard to store the whole packet streams even just with the header portion [62].

Second, the volume of data is relatively limited in most traditional database applications. This is unlike streaming data that can be potentially infinite in terms of its size, because new streaming data keeps arriving.

Third, multiple passes of data is possible in most traditional database applications. Since it is assumed that all data is stored on disks, DBMSs can load the data as many times as they want to evaluate a query. In data stream applications, the full data stream can be only processed once; queries can be answered either based on current stream information in memory or a partial archived stream or sketch on disks.

Fourth, random access to the data is possible in most traditional database applications. Since all data is available on disks, DBMS can access any part of the data to answer a query, whereas in data stream applications, the data stream can only be accessed sequentially. The order of which data is processed is not controllable and is determined by the source generating the stream. When a partial stream is archived, it is possible to gain random access to the partial data. But it is only useful in the offline query case, and the query answer may not be precise.

Fifth, precise query answers are usually required in most traditional database applications. Again, since all data is available, query answers in most traditional database applications are usually precise. In comparison, due to the high arrival rate of streaming data, some queries are hard to be answered precisely and timely. For example, if users are interested in finding the number of distinct elements in the stream, for a precise answer a large amount of fast memory needs to be used to store all the distinct elements. Since the stream can be potentially unbounded, the available memory may not always be sufficient to evaluate this query effectively.

Finally, in most traditional database applications, time information (transaction or valid time) is not kept or is not important. If time needs to be included, it is usually organized as a separate attribute, and many queries do not include time. However, in data stream applications, timing information is much more important because data keeps arriving and changing. Without specifying the time, the query would be ambiguous and incomplete.

1.3 Data Stream Model

For the kind of applications discussed here, a data stream can be modeled as a sequence of records $S = e_1 \ldots e_N$ ordered by their arrival time $t_1 \ldots t_N$, where N is the size of the stream, and e_N is the most recent record seen so far. Each record may contain one or more attributes such as record identifier, arrival time, record size, etc.

1.3.1 Tuple-based and Time-based Stream

We can further classify data streams into *tuple-based* and *time-based*. In tuple-based model, the record arrival time is represented simply by the subscripts of the record. For example, the arrival time of e_N is N. In time-based model, the record arrival time is represented by a separate timestamp attribute.

1.3.2 One-time Query and Continuous Query

Data stream queries can be classified into *one-time queries* and *continuous queries* [9]. One-time queries are issued occasionally like traditional DBMS queries, while continuous queries are issued and evaluated frequently and regularly with the arrival of streaming data. For example, in tuple-based query model, a continuous query should be evaluated whenever a record arrives; in time-based query model, the query should be evaluated after every fixed time interval.

1.3.3 Query Windows

Based on the temporal span of the queries, we can classify query windows into *landmark windows*, *sliding windows* and *damped windows*[121]. In the landmark window model, user queries are answered based on the data arrived between a particular timepoint (called landmark) and present. The landmark does not shift when the current timepoint keeps moving towards the future, unless a new landmark is redefined explicitly later. In the sliding window model, user queries are answered based on the data arrived between a starting timepoint and the current timepoint. In contrast to the landmark window model, the starting timepoint shifts accordingly when the current timepoint keeps moving towards the future, and the time span between them is always fixed. In the damped window model, the data under consideration are weighted differently. Users can specify a data importance function which affects the query results. For example, one can specify an exponential decay function indicating that recent data is much more important than the old ones. Damped windows are considered in some past work [93, 103, 38] where applications are also provided.

1.4 Data Stream Research Sketch

In this section, a brief summary of current data stream research is provided. The work can be classified into two categories: data stream systems and data stream algorithms.

1.4.1 Data Stream Systems

Some of the data stream research focus on building general-purpose data stream systems, while others specialize in a particular application domain. Examples of general-purpose data stream systems are STREAM [9, 6], AURORA [27, 2, 13], and TelegraphCQ [28, 87]. Examples of specialized data stream systems include Gigascope [51, 52], NIAGARA [101, 37], and Tribeca [110].

STREAM. Mainly motivated by the Internet traffic monitoring and measurement, the STREAM project at the Stanford University aims at implementing a general-purpose data stream system, which may be applied to financial data analyses, telecommunication data management and click stream monitoring as well as IP traffic monitoring. STREAM focuses on online continuous queries, processing data streams in real-time and producing continuous output streams. In addition to generating a live output stream mainly based on data in memory, STREAM can archive the input streams on disks for offline analyses and sometimes stores the output results as relations on disks. Furthermore, the real-time input streams can be combined with static stored relations to generate the output. Registered continuous queries in STREAM are expressed using a declarative language which is an extension of SQL.

Aurora. The joint project, Aurora [27], from the Brandeis University, Brown University and MIT is another example of general-purpose data stream systems. Based on user-specified queries (continuous or ad-hoc), the input streams are processed by a centralized stream engine, and the output streams are provided to applications. Within the stream engine, there are a set of operators ("boxes" in their terminology), including "windowed" operators, Filter, Drop, Map, GroupBy and Join operators. In addition to on-the-fly processing of the input streams, Aurora can archive input streams to disks especially for answering ad-hoc queries. Another feature of Aurora is that an application can specify Quality of Service (QoS), and the system can make run-time optimizations to provide the service. For example, when the system is overloaded, load shedding can be done based on user-provided QoS description rather than randomly dropping data.

Aurora* [40] is the distributed version of Aurora. More recently, Borealis [1] supersedes Aurora*.

TelegraphCQ. TelegraphCQ is an extension of Telegraph [108] for continuous queries. The features of TelegraphCQ are as follows. First, different types of queries are considered, including snapshot queries (one-time queries), landmark window queries and sliding window queries. Second, disks are used to archive historical data. Third, QoS is considered in managing the limited resources.

AT&T Gigascope. Gigascope [51, 52] is a data stream system dedicated to Internet monitoring. More precisely, Gigascope can be used for TCP, IP or higher level traffic measurements and identifications.

Tribeca. Tribeca [110] is another data stream system designed for network traffic analysis. Besides basic operators such as qualifications (filters), projections and aggregates, Tribeca also provides demultiplexing and remultiplexing which partitions and combines data streams. Another features of Tribeca is the support of secondary indices on sorted fields of the data stream records (e.g. timestamp) since the network traffic can be stored on tapes and analyzed in an offline manner.

Niagara. Niagara [101, 37] is a data stream system focusing on XML documents. In collaboration with an XML search engine, Niagara processes XML document streams from the search engine, possibly with different arrival rates.

1.4.2 Data Stream Algorithms

In the previous subsection, we briefly describe some of the data stream systems and their features. Another research direction is on data stream algorithms. This is a subject that has drawn great interest not only from the database community, but also from the theory community. Since there are too many papers on this area, we just review those closely related to our work in the next chapter. More work in this area can be found in the surveys by Babcock et al. [9] and Muthukrishnan [100].

1.5 Thesis Overview

This section briefly discusses the scope of this thesis and its contributions.

1.5.1 Thesis Scope

In this thesis, we focus on a particular class of data stream algorithms for approximately answering a few important frequency related queries: membership queries, iceberg queries, join/self-join size estimations, similarity join/self-join size estimation and selectivity estimation for similarity searches. The main techniques we explore are sketching and sampling; both techniques are based on probabilistic analysis, and can be considered as lossy compression of streams to answer particular queries.

Sketching. Sketching is a summarization technique where each data stream element contributes to the summary. This method usually takes advantage of hash functions to transform stream elements and construct a summary. For example, an IP packet header consisting of several fields can be transformed into a bit in the summary, thus saving space. Of course, certain information such as IP source and destination addresses are lost. For certain queries where those lost information is not relevant or important, sketching techniques can be quite useful.

Sampling. Sampling is also a summarization technique where a subset of the elements of the original stream is obtained. Sampled stream elements or records are exactly the same as the ones in the original stream, and not every stream element contributes to the sample. One nice property of sampling is that it keeps the entire record in the sample, which can be useful for different purposes. Based on the queries to be answered, there can be many different sampling strategies.

More details about sketching and sampling are discussed in Chapter 2.

1.5.2 Contributions

A summary of the contributions of this thesis is as follows:

- For approximate membership query, this thesis extends and generalizes a well known sketching technique, Bloom filters, to be applicable to data stream scenario, and accordingly, present a novel algorithm which dynamically updates the sketch to represent recent data in a data stream. Some important properties of this new sketching technique are investigated and proved. Experiments on both real-world and synthetic data show that this technique improves upon some of the best known algorithms.

- For frequency related queries, this thesis describes a new sketching technique CMM, whose estimation accuracy is very similar to a well known sketching technique, AMS sketch, but CMM is more flexible and powerful because the new technique can be combined with another query answering method, Count-min, using exactly the same sketch. Analytical and experimental results are both provided, showing CMM is indeed a more flexible and powerful method.
- For similarity join and self-join size estimations and similarity search selectivity estimations, this thesis presents efficient one-pass algorithms, approximately answering the queries with provable accuracy and high probability using only small amount of space. The key idea is to map a stream of multi-dimensional objects into one-dimensional data streams via sampling such that efficient data stream sketches can be used for the transformed stream.

Chapter 2

Sketching and Sampling

This chapter gives background knowledge about two data reduction techniques, namely, sketching and sampling. A review of more data reduction techniques including singular value decomposition (SVD), wavelets, regression, histograms and so on can be found in the New Jersey data reduction report [15].

2.1 Sketching

In general, sketching techniques use hash functions to map each data stream element into a hash value, and update a space efficient data structure (sometimes called a sketch) based on the hash value. After processing the whole stream, certain queries can be answered based on the sketch content. Usually all stream elements contribute to the sketch, unlike in the sampling techniques. Note that sketching is not a strictly defined term, and it may have different interpretations in other places.

2.1.1 Bloom Filters and their extensions

Bloom filters. Bloom [17] proposes a synopsis data structure, known as the Bloom filter, to approximately answer membership queries. A Bloom filter, BF, is a bit array of size m, all of which are initially set to 0. For each element, K bits in BF are set to 1 by a set of hash functions $\{h_1(x), \ldots, h_K(x)\}$, all of which are assumed to be independent perfect random hash functions. It is possible that one bit in BF is set multiple times, while only the first setting operation changes 0 into 1, and the rest has no effect on that bit. To know whether a newly arrived element x_i has been seen before, we can check the bits $\{h_1(x_i), \ldots, h_K(x_i)\}$. If any one of these bits is zero, with 100% confidence we know x_i is a distinct element. Otherwise, it is regarded as a duplicate with a certain probability of error. An error may occur because it is likely that the cells $\{h_1(x_i), \ldots, h_K(x_i)\}$ are set before by elements other than x_i.

The probability of a false positive (*false positive rate*) $FP = (1 - p)^K$, where $p = (1 - 1/m)^{Kn}$ is the probability that a particular cell is still zero after seeing n distinct elements. It is shown that when the number of hash functions $K = ln(2)(m/n)$, this probability will be minimized to $(1/2)^{ln(2)(m/n)}$, where m is the number of bits in BF and n is the number of distinct elements seen so far [98].

Counting Bloom filters. Elements can be inserted into a BF, but cannot be deleted. To handle deletions, Fan et al. [66] extend BF to Counting Bloom Filters (CBF), in which each bit of a Bloom filter is changed to a counter. Whenever an element is inserted, all counters the element is mapped to are incremented by one. To delete the element, those counters are decremented by one. Even though CBF is originally proposed for membership queries, it can be used to answer multiplicity queries as well.

Spectral Bloom filters. Cohen and Matias [42] propose Spectral Bloom Filters (SBF) to answer multiplicity queries. An SBF is a 1-dimensional array of counters, initially all set to 0. To insert an element into the SBF, k hash functions are used to pick k counters uniformly at random, and those counters are incremented by 1. To answer a query, the k counters the query element has touched are checked, and the minimum value of those k counters is returned as the approximate query answer.

To increase the accuracy, they propose two independent (but incompatible) heuristics: Minimal Increase (MI) and Recurring Minimum (RM). To insert an element x into the SBF in the MI heuristic, only the minimum counter/counters rather than all of the counters x touches is increased by 1. This heuristics decreases the error because it makes the counters increase slower. However, the error reduction depends on data distribution and the order of element insertions. Therefore, analyses become hard and are given in [42] only in the case that element frequencies are uniformly distributed. Also MI does not support element deletions, unlike the basic SBF.

The other heuristic, RM, uses a secondary data structure, which is the same as the main data structure, to store certain elements that have higher chance of being wrongly estimated. The following observation is used to find such elements: of the counters an element touches, if there are multiple counters that share the minimum value, this element has less chance of being wrongly estimated; otherwise, RM stores this element in a separate secondary data structure. RM is less accurate than MI, but RM supports element deletions.

Count-min Sketches. A count-min sketch $CM[i,j]$ is a 2-dimensional array of counters, with d (sketch depth) rows and w (sketch width) columns. All counters are initially set to 0. To insert an element x into the sketch, d hash functions $h_i(x) \in \{0,\ldots,w-1\}$ with $i = 0,\ldots,d-1$, picked uniformly at random, are used to determine which counters to be updated. For each row i, counter $CM[i,h_i(x)]$ is incremented by 1. The procedure to delete an element x is similar: for each row i, counter $CM[i,h_i(x)]$ is decremented by 1.

To find the number of occurrences of an element x, all the d counters that x has touched, i.e. $CM[i,h_i(x)]$ with $(i = 0,\ldots,d-1)$, are checked, and the minimum counter value is returned as the estimated frequency of x. Clearly, the estimate is an upper bound of the true frequency.

Having a count-min sketch, the self-join size can be estimated as follows: For each row i of the sketch, sum up the square of each counter value in that row, and return the minimum sum of all d rows as the estimate. That is, the estimate $\hat{F}_2 = min\{sum_{j=0}^{w-1}(CM[i,j])^2, i = 0, \ldots, d-1\}$. This estimate is an upper bound of the true value as well.

Distinct counting using Bloom filters. In addition to aforementioned membership queries, multiplicity queries, self-join size, Bloom filters can be used to approximate the number of distinct elements in a data stream. By counting the fraction of zero bits in a Bloom filter, Whang et al. [119] estimate the number of distinct elements using space linear in the number of distinct elements, and constant time for updating each element. To have a reasonable estimation accuracy, one has to know a rough estimate of the number of distincts so as to allocate a proper amount of space handling the stream. However, this may not always be possible in practice. To tackle this, Estan et al. [64] propose multi-resolution Bloom filters where stream elements are partitioned into multiple groups based on their hash values. The number of distinct elements hashed to different groups are exponentially decreasing. For each group, they use a fixed size Bloom filter to represent the elements as in a regular Bloom filter. In the end, they estimate the final results based on those Bloom filters which are not full (i.e. there are a certain fraction of zeros in the bitmap).

Space-Code Bloom filters (SCBF) and flow size distribution estimation using Bloom filters. Kumar et al. [89] extend Bloom filters to estimate element multiplicities. For each Bloom filter, they use multiple groups of hash functions instead of one group as in the regular Bloom filters. When inserting an element into the Bloom filter, they pick a group of hash functions uniformly at random and use them to hash the element into the SCBF as in a regular Bloom filter. By checking the number of bits set by an element after the whole data stream has been processed, one can estimate the multiplicity of the element. However, if the multiplicity of an element is more than a threshold, this approach will not work anymore because after all bits that can be possibly touched by the element have been set, any more occurrences will not change the SCBF. Thus, a multi-resolution SCBF which consists of multiple SCBFs is proposed by Kumar et al. [89]. Whenever an element arrives, it is hashed to each of the multiple SCBFs with a different probability. Accordingly, only a fraction of the stream elements (random sample) will touch each SCBF. By probing the SCBF with a high sampling rate, the multiplicities of frequent elements can also be estimated. Since in their paper there are no comparisons with other methods such as Fast-AGMS sketching [31], which will be discussed later in this chapter, it is not clear how well this technique performs.

Kumar et al. [88] also use Counting Bloom filters to estimate flow size distributions, where the flow size can be considered as the multiplicity of an element. In the network community, a flow usually refers to a number of IP packets sharing certain common properties, e.g. source and destination IP addresses. Thus, estimating the flow size distribution is to find out the number of distinct elements which occur a certain number of times. Kumar et al. summarize the stream using a Counting Bloom filter. By guessing the combinations of each counter value, they find the combinations maximizing the likelihood of the counter value distribution. They compare their approach with the results from a random sampling [58], but it is not clear how their approach performs compared with distinct sampling [54, 50].

2.1.2 FM Sketches and their extensions

Another well-known sketching technique was proposed by Flajolet and Martin [67], to approximate the number of distinct elements in a data stream.

FM Sketching. The basic idea of FM sketching is as follows. The sketch consists of a set of bitmaps. Whenever an element arrives, it is hashed to one bit in a bitmap, and the probability that the element is hashed to a more significant bit is exponentially decreasing. For example, the probability that an element touches the least significant bit is $1/2$, and the probability of touching the next bit is $1/4$, and the probability of touching the third least significant bit is $1/8$. One way of implementing the hash function is to hash the element uniformly at random to a binary string with the same size as the bitmap; then find the least significant 1 bit and return the position of that bit as the hash value. One advantage of this technique is the space efficiency. However, it may not be suitable for real-time applications where per element processing time is critical because its per element processing time is proportional to the number of bitmaps used and thus proportional to the size of allocated space.

AMS variation. In Flajolet and Martin's paper, the authors assume the hash function is truly random; since it is not known how it can be implemented in practice, Alon et al. [4] modified FM sketches to the case that only pairwise independent hash functions are needed. Another change is that they only keep track of the most significant position in the sketch that has been touched after processing the whole stream rather than the whole sketch. But this approach does not improve FM sketch's time efficiency.

Min hash value approach. Bar-Yossef et al. [14] extend the idea of Alon et al. and propose another algorithm for counting distinct elements. The algorithm works as follows: whenever an element arrives, hash it to a large universe (to avoid hash collisions) uniformly at random; meanwhile maintain the k smallest hash values seen so far; after processing the whole stream, one can estimate the number of distinct elements based on the largest value of the k hash values stored. This method has a space bound similar to the above two techniques, and has a much better per element processing time, which is $O(log(1/\epsilon) \cdot loglogm \cdot log(1/\delta) \cdot logm)$, where ϵ is the relative error, m is the universe size, and δ is the confidence probability. Although this method is both space and time efficient, it is not clear how accurate it would perform in practice especially compared with a Bloom filter based approach.

Space bounds. One important property of all above algorithms is that they all need only $1/\epsilon^2(O(logn) + O(logm))$ bits space to bound the relative error within ϵ factor and with a constant probability, where m and n are the universe of stream elements and stream size respectively. It means that a small amount of space is enough to accurately estimate the number of distinct elements in a massive data stream. Moreover, Woodruff [120] showed that $1/\epsilon^2$ is the space lower bound needed for any approximation algorithm in terms of ϵ. Note that ϵ must be no less than $1/\sqrt{m}$ since otherwise a straightforward deterministic algorithm is applicable when the space is large enough. This lower bound is important because it indicates that the space complexity of above algorithms is already optimal in terms of ϵ.

2.1.3 AMS Sketches and extensions

AMS sketches. Alon et al. [4] proposed sketching techniques, to approximate k-th frequency moment of a data stream. The k-th frequency moment, F_k, is $\sum_{i \in D} f_i^k$ where D is the universe or the domain from which the element values are drawn, and f_i is the frequency of value i in the stream. In particular, the well-known sketching technique, referred to as AMS sketch, is developed for approximating F_2. The basic idea is as follows: for each data stream element, use a 4-wise independent hash function to hash it into either -1 or 1, and store the sum of the hash values of all stream elements into a counter; this is one instance of the sketch; by using a set of independent hash functions, one can obtain a set of counters. After processing the whole stream, the median of all these counters turns out to be an unbiased estimate for F_2.

Fast-AGMS Sketches. Based on the AMS sketching technique[4], Charikar, Chen and Farach-Colton [31] propose Count-sketches to estimate element multiplicities. The same sketches are also called Fast-AGMS sketches [46] in self-join size estimation scenarios. For the ease of presentation, we only use the term Fast-AGMS sketches, to refer to this data structure in the rest of the paper.

The Fast-AGMS sketches are organized as a 2-dimensional array of counters. To insert an element into the sketch, for each row of the sketch, a hash function is used to determine which counter should be updated according to the hash value of the element, and another independent hash function maps the element to either -1 or 1 uniformly at random, indicating the value to be added to the counter. To delete an element from the sketch, based on the same hash functions either -1 or 1 is deducted from the counters the element is hashed to.

To check the multiplicity of a query element, for each row of the sketch, map the element into a counter and a value (either -1 or 1), using the same two hash functions as in the sketch construction process. Obtain the product between the hash value (-1 or 1) and the value of the counter the element is mapped to, then report the median of those products from all rows as the multiplicity estimate. This estimate is shown [31] to be unbiased.

To estimate the self-join size, for each row of the sketch, sum up the squares of all counter values, and return the median of those sums from all rows as the self-join size estimate. Again, this is also an unbiased estimate.

2.2 Sampling

Sampling has been used in many areas and has been studied for many years. As a data reduction technique, sampling has also a wide range of applications including traditional database systems [102]. Here we just focus on using sampling to answer frequency related queries for streaming data.

2.2.1 Random Sampling

Among different types of sampling, random sampling probably is the most widely studied topic since it has lots of applications and can be used to answer different types of queries. For example, a random sample can be used to answer queries on the number of distinct elements [30], the total number of elements, average data element value and so on.

In the database area, several work have been focusing on how to maintain a random sample. Vitter [115] studied the problem of obtaining a fixed size random sample for a data set in one pass. Babcock et al. [10] proposed algorithms to maintain a random sample of streaming data within a sliding window. Jermaine et al. [82] studied how to efficiently maintain a large random sample on disks.

Random samples can be used to answer many types of queries. However, for many frequency related queries, the results from a random sample may not be accurate. For example, Charikar et al. [10] showed negative results for counting distinct elements using random sampling. For inverse distribution estimation [50], where the goal is to find the numbers of distinct elements with different frequencies (e.g. the number of elements only appear once in the stream), it is not hard to see that random sampling is not proper since high frequency elements will dominate the sample, and a large fraction of low frequency elements will not be sampled. In many of those scenarios, distinct sampling may be a better option.

2.2.2 Distinct Sampling

Another sampling technique that has many applications is distinct sampling. The main difference between random sampling and distinct sampling is that in distinct sampling, once a distinct element is included in the sample, all its duplicates will be sampled. This is because the sampling process often involves hashing the streaming element into a hash value; based on the hash value, the element either is kept in the sample or discarded. Thus, the chance that an element stays in the sample is irrelevant to its frequency unlike in the random sampling case.

Gibbons [71] studied the problem of finding the number of distinct values of one attribute while some other attributes of the records satisfy certain predicates. An example query can be: "How many distinct source IP addresses have sent TCP packets in the IP packet stream?" In this query, the goal is to find the number of distinct source IP addresses, but only those packets whose protocol type is TCP are under consideration, so sketching methods are not applicable in this case.

2.2.3 Count sample and sticky sample

Another type of sampling technique in the data streaming area is count sampling or sticky sampling, which focus on answering frequent element queries. Gibbons and Matias [72] proposed count sampling technique, where the main idea is as follows. Whenever an streaming element arrives, they check the current sample; if the element is found, increment its frequency counter by 1;

if not found, add the element to the current sample with probability p. When the sample size bound is reached, for each element in the sample, they flip biased coins until a head appears, and they decrement the number of coin tosses from the frequency counter of that element. Meanwhile, they decrease the probability p to p' to set a smaller probability of inserting a new element into the sample. Manku and Motwani [94] used a similar idea to answer iceberg queries rather than top-k frequent element queries, with different detailed analysis.

Chapter 3

Approximate Membership Query Processing

In this chpater, we propose *Stable Bloom Filter(SBF)* [56], which extends and generalizes the regular Bloom filter, and accordingly, a novel algorithm which dynamically updates the sketch to represent recent data. We find and prove the stable properties of an SBF including stability, exponential convergence rate and monotonicity, based on which we show that using constant space the chance of a false positive can be bounded to a constant independent of the stream size, and this constant is explicitly derived. Furthermore, we show that the processing time of SBF for each element in the stream is also constant independent of the stream size. To make our algorithm readily applicable in practice, we provide detailed discussions of the parameter setting issues both in theory and in experiments. And we compare our method to alternative methods using both real and synthetic data. The result shows that our method is superior in terms of both accuracy and time efficiency when a fixed small space and an acceptable false positive rate are given.

Section 3.1 describes the membership query and its importance. In Section 3.2, we present the problem statement and some background on existing approaches. Our solution is presented and discussed in Section 3.3. In Section 3.4, we discuss how our algorithm can be used in practice. In Section 3.5, we verify our theoretical findings, experimentally evaluate our method and report the results of our comparisons to those from alternative methods. Related work is reviewed in Section 3.6, and conclusions and future work are discussed in Section 3.7.

3.1 Approximate Membership Query

A membership query is to answer if a given element belongs to a set or not. From another angle, this query can be described as "if the given element has a non-zero frequency or multiplicity". It is also known as duplicate detection or duplicate elimination problem. Eliminating duplicates is an important operation in traditional query processing, and many algorithms have been developed [69]. A common characteristic of these algorithms is the underlying assumption that the whole data set is stored and can be accessed if needed. Thus, multiple passes over data are possible, which is the case in a traditional database scenario. However, this assumption does not hold in the streaming applications, which are becoming increasingly important. Consequently, detecting duplicates precisely is not always possible. Instead, it may suffice to identify duplicates with errors which will be discussed later.

While it is useful to have duplicate elimination in a Data Stream Management System (DSMS)[9], some new properties of these systems make the duplicate detection problem more challenging and to some degree different from the one in a traditional DBMS. First, the timely response property of data stream applications requires the system to respond quickly in real-time. There is no choice but to store the data in limited main memory rather than in huge secondary storage. Sometimes even main memory is not fast enough. For example, for network traffic measurement and accounting, ordinary memory (DRAM) is too slow to process each IP packet in time, and fast memory (on-chip SRAM) is small and expensive [63, 12].

Second, the potentially unbounded property of data streams indicates that it is not possible to store the whole stream in a limited space. As a result, exact duplicate detection is infeasible in such data stream applications.

On the other hand, there are cases where efficiency is more important than accuracy, and therefore a quick answer with an allowable error rate is better than a precise one that is slow. Sometimes there is no way to have a precise answer at all. Therefore, load shedding is an important topic in data stream system research [111, 11]. Next, we provide some motivating examples.

3.1.1 Motivating Examples

URL Crawling. Search engines regularly crawl the Web to enlarge their collections of Web pages. Given the URL of a page, which is often extracted from the content of a crawled page, a search engine must probe its archive to find out if the URL is in the engine collection and if the fetching of the URL can be avoided [24, 78].

One way to solve the problem is to store all crawled URLs in main memory and search for a newly encountered URL in it. However, the set of URLs can be too large to fit in memory. Partially storing URLs in the secondary storage is also not perfect because of the large volume of searches that is expected to be performed within a time unit.

In practice, detecting duplicates precisely may not be indispensable. The consequence of an imprecise duplicate detection is that some already-crawled pages will be crawled again, or some new URLs which should be crawled are missed. The first kind of error may lead the crawler to do some redundant crawling. This may not have a great influence on performance as long as the error rate is not high. For the second kind of errors, since a search engine can archive only a small portion of the entire web, a small miss rate is usually acceptable. In addition, if a missed URL refers to a high quality page, it is quite likely that the URL will be listed in the content of more than one crawled page, and there is less chance of missing it again. The solution adopted by the Internet Archive crawler introduces the second kind of errors[78].

Selecting distinct IP addresses. In network monitoring and accounting, it is often important to understand the traffic and users on the network [79]. The following two queries, for example, may be interesting to network monitors: who are the users on the network within past one hour? Where do they go? The query may be written as:

SELECT DISTINCT source ip, destination ip

FROM ip_packetsstream

WITHIN PAST 1 hour

The result could be helpful for further analyzing the user profiles, interests and the network traffic. Because of the high throughput of Internet routers and limited amount of fast memory, currently it is hard to capture per packet information precisely, and sampling is often used as a compromise [61]. We are not aware of any work using sampling for this duplicate elimination problem.

Duplicate detection in click streams. Recently, Metwally et al. propose another application for approximate duplicate detection in a streaming environment [96]. In a Web advertising scenario, advertisers pay web site publishers for clicks on their advertises (or links). For the sake of profit, it is possible that a publisher fakes some clicks (using scripts), hence a third party, called an advertising commissioner, may want to detect those false clicks by monitoring duplicate user IDs. We discuss more about their work in the related work section.

3.2 Preliminaries

This section presents the problem statement in a data stream model and some possible solutions.

3.2.1 Problem Statement

We consider a data stream as a sequence of numbers, denoted by $S_N = x_1, \ldots, x_i, \ldots, x_N$, where N is the size of the stream. The value of N can be infinite, which means that the stream is not bounded. In general, a stream can be a sequence of records, but it is not hard to transform each record to a number (e.g., using hashing or fingerprinting) and use this stream model.

Our problem can be stated as follows: given a data stream S_N and a certain amount of space, M, estimate whether each element x_i in S_N appears in $x_1, \ldots, x_{i-1}$ or not. Since our assumption is that M is not large enough to store all distinct elements in $x_1, \ldots, x_{i-1}$, there is no way to solve the problem precisely. Our goal is to approximate the answer and minimize the number of errors, including both false positives and false negatives , where a false positive is a distinct element wrongly reported as duplicate, and a false negative is a duplicate element wrongly reported as distinct.

To address this problem, we examine two techniques that have been previously used in different contexts.

3.2.2 The Buffering Method

A straightforward solution is to allocate a buffer and fill the buffer with enough elements of the stream. For each new element, the buffer can be checked, and the element may be identified as distinct if it is not found in the buffer, and duplicate otherwise. When the buffer is full, a newly arrived element may evict another element out of the buffer before it is stored. There are many replacement policies for choosing an element to be dropped out (e.g. [69]). Clearly, buffering introduces no false positives. We will use this method in our experiments and compare its performance to that of our method.

3.3 Stable Bloom Filters

The Bloom filter is shown to be useful for representing the presence of a set of elements and answering membership queries, provided that a proper amount of space is allocated according to the number of distinct elements in the set.

3.3.1 The Challenge to Bloom Filters

However, in many data stream applications, the allocated space is rather small compared to the size of the stream. When more and more elements arrive, the fraction of zeros in the Bloom filter will decrease continuously, and the false positive rate will increase accordingly, finally reaching the limit, 1, where every distinct element will be reported as a duplicate, indicating that the Bloom filter is useless.

Our general solution is to avoid the state where the Bloom filter is full by evicting some "elements" from it before the error rate reaches a predefined threshold. This is similar to the replacement operation in the buffering method, in which there are several possible policies for choosing a past element to drop. In many real world data stream applications, often the recent data is more important than the older data [41, 104]. However, for the regular Bloom filter, there is no way to distinguish the recent elements from the past ones, since no time information is kept. Accordingly, we add a random deletion operation into the Bloom filter so that it does not exceed its capacity in a data stream scenario.

3.3.2 Our Approach

To solve this problem, we introduce the *Stable Bloom Filter*, an extension of the regular Bloom filter.

Definition 1 (Stable Bloom Filter (SBF)). *An SBF is defined as an array of integer* $SBF[1], \ldots, SBF[m]$ *whose minimum value is* 0 *and maximum value is* Max*. The update process follows Algorithm 1. Each element of the array* *is allocated d bits; the relation between Max and d is then $Max = 2^d - 1$. Compared to bits in a regular Bloom filter, each element of the SBF is called a* **cell**.

Concretely speaking, we change bits in the regular Bloom filter into cells, each consisting of one or more bits. The initial value of the cells is still zero. Each newly arrived element in the stream is mapped to K cells by uniform and independent hash functions. As in a regular Bloom filter, we can check if a new element is duplicate or not by probing whether all the cells the element is hashed to are non-zero. This is the duplicate detection process.

After detecting duplicates, we need to update the SBF. We first randomly decrement P cells by 1 so as to make room for fresh elements; we then set the same K cells as in the detection process to Max. Our symbol list is shown in Table 3.1, and the detailed algorithm is described in Algorithm 1.

Algorithm 1: Approximately Detect Duplicates using SBF

```
Data: A sequence of numbers $S = x_1, \ldots, x_i, \ldots, x_N$.
Result: A sequence of "Yes/No" corresponding to each input number.
begin
    initialize SBF[1]... SBF[m] = 0
    for each $x_i \in S$ do
        Probe $K$ cells $SBF[h_1(x_i)] \ldots SBF[h_K(x_i)]$
        if none of the above $K$ cells is 0 then
            DuplicateFlag = "Yes"
        else
            DuplicateFlag = "No"
        Select $P$ different cells uniformly at random
        $SBF[j_1] \ldots SBF[j_P], P \in \{1, \ldots, m\}$
        for each cell $SBF[j] \in \{SBF[j_1], \ldots, SBF[j_P]\}$ do
            if $SBF[j] \geq 1$ then
                $SBF[j] = SBF[j] - 1$
        for each cell $\in \{SBF[h_1(x_i)], \ldots, SBF[h_K(x_i)\}$ do
            $SBF[h(x_i)] = Max$
        Output DuplicateFlag
end
```

Table 3.1: The Symbol List

Symbols	Meanings
N	Number of elements in the input stream
M	Total space available in bits
m	Number of cells in the SBF
Max	The value a cell is set to
d	Number of bits allocated per cell
K	Number of hash functions
k	The probability that a cell is set in each iteration
P	Number of cells we pick to decrement by 1 in each iteration
p	The probability that a cell is picked to be decremented by 1 in each iteration
h_i	The ith hash function

3.3.3 The Stable Property

Based on the algorithm, we find an important property of SBF both in theory and in experiments: after a number of iterations, the fraction of zeros in the SBF will become fixed no matter what parameters we set at the beginning.

We call this the *stable property* of SBF and deem it important to our problem because the false positive rate is dependent on the fraction of zeros in SBF.

Theorem 1. *Given an SBF with m cells, if in each iteration, a cell is decremented by 1 with a probability p and set to Max with a probability k, the probability that the cell becomes zero after N iterations is a constant, provided that N is large enough, i.e.*

$$\lim_{N\to\infty} Pr(SBF_N = 0)$$

exists, where SBF_N is the value of the cell at the end of iteration N .

In our formal discussion, we assume that the underlying distribution of the input data does not change over time. Our experiments on the real world data show that this is not a very strong assumption, and the experimental results verify our theory.

Proof. Within each iteration, there are three operations: detecting duplicates, decreasing cell values and setting cells to Max. Since the first operation does not change the values of the cells, we just focus on the other two operations.

Within the process of iterations from 1 to N, the cell could be set $0, \ldots, (N-1)$ or even N times. Since the newest setting operation clears the impact of any previous operations, we can just focus on the process after the newest setting operation.

Let A_l denote the event that within the N iterations the most recent setting operation applied to the cell occured at iteration $N-l$, which means that no setting happened within the most recent l iterations (i.e. from iteration $N-l+1$ to iteration N, $l<N$), and let $\bar{A}_N$ denotes the event that the cell has never been set within the whole N iterations. Hence, the probability that the cell is zero after N iterations is as follows:

$$\begin{aligned} Pr(SBF_N = 0) = &\sum_{l=Max}^{N-1} [Pr(SBF_N = 0 \mid A_l) Pr(A_l)] \\ &+ Pr(SBF_N = 0 \mid \bar{A}_N) Pr(\bar{A}_N), \end{aligned} \tag{3.1}$$

where

$$Pr(SBF_N = 0 \mid A_l) = \sum_{j=Max}^{l} \binom{l}{j} p^j (1-p)^{l-j} \tag{3.2}$$

$$Pr(A_l) = (1-k)^l k \tag{3.3}$$

$$Pr(SBF_N = 0 \mid \bar{A}_N) = 1 \tag{3.4}$$

$$Pr(\bar{A}_N) = (1-k)^N. \tag{3.5}$$

We have Eq 3.2 because during those l iterations, there is no setting operation, and the cell becomes zero if and only if it is decremented by 1 no less than Max times. Clearly when $l < Max$, the cell is impossible to be decreased to 0, and

$l = N$ means $\bar{A}_N$ happens, so we just consider the cases of $Max \le l \le (N-1)$ in Eq 3.2. When $\bar{A}_N$ happens, the cell is 0 with a probability 1 because the initial value of the cell is 0 and it has never been set, therefore we have Eq 3.4. Having the above equations, we can prove that $\lim_{N\to\infty} Pr(SBF_N = 0)$ exists.

From Eq 3.4 and Eq 3.5 we can clearly see the limit of the second part of $Pr(SBF_N = 0)$ (when N goes to infinity)

$$Pr(SBF_N = 0 \mid \bar{A}_N) * Pr(\bar{A}_N)$$

is 0. So we just need to focus on the first part of Eq 3.1, i.e.

$$\sum_{l=Max}^{N-1} (Pr(SBF_N = 0 \mid A_l) * Pr(A_l)),$$

which is denoted by $Pr_{Part1}(0)$ in the rest of this proof. Now let us look at the first part of $Pr_{Part1}(0)$:

$$\begin{aligned} Pr(SBF_N = 0 \mid A_l) &= \sum_{j=Max}^{l} \binom{l}{j} p^j (1-p)^{l-j} \\ &= 1 - \sum_{j=0}^{Max-1} \binom{l}{j} p^j (1-p)^{l-j} \le 1 - \binom{l}{0} p^0 (1-p)^{l-0} \\ &= 1 - (1-p)^l \end{aligned}$$

$$\begin{aligned}\therefore Pr_{Part1}(0) &\leq \sum_{l=Max}^{N-1}(1-(1-p)^l)(1-k)^l k\\ &= \sum_{l=Max}^{N-1}(k(1-k)^l) - \sum_{l=Max}^{N-1}(k(1-k)^l(1-p)^l)\\ &= (1-k)^{Max} - (1-k)^N\\ &- \frac{k}{k+p-kp}((1-k)^{Max}(1-p)^{Max} -\\ &(1-k)^N(1-p)^N)\end{aligned}$$

$$\begin{aligned}&\therefore \lim_{N\to\infty} Pr_{Part1}(0)\\ &\leq (1-k)^{Max} - \frac{k}{k+p-kp}((1-k)^{Max}(1-p)^{Max}\end{aligned}$$

$\because Pr_{Part1(0)}$ monotonically increases when $N \in [Max, \infty)$ increases $\therefore \lim_{N\to\infty} Pr_{Part1}(0)$ exists. $\because$ The limit of the second part of $Pr(SBF_N = 0)$ also exists. $\therefore$ The limit of $Pr(SBF_N = 0)$ exists. □

Having Theorem 1, now we can prove our stable property statement.

Corollary 1 (Stable Property). *The expected fraction of zeros in an SBF after N iterations is a constant, provided that N is large enough.*

Proof. In each iteration, each cell of the SBF has a certain probability of being set to Max by the element hashed to that cell. Since the underlying distribution of the input data does not change, the probability that a particular element appears in each iteration is fixed. Therefore, the probability of each cell being set is fixed.

Meanwhile, the probability that an arbitrary cell is decremented by 1 is also a constant. According to Theorem 1, the probabilities of all cells in the SBF becoming 0 after N iterations are constants, provided that N is large enough. Therefore, the expected fraction of 0 in an SBF after N iterations is a constant, provided that N is large enough. □

Now we know an SBF converges. In fact this convergence is like the process that a buffer is filled by items continually. SBF is stable means that its maximum capacity is reached, similar to the case that a buffer is full of items. Another important property is the convergence rate.

Corollary 2 (Convergence Rate). *The expected fraction of* 0*s in the SBF converges at an exponential rate.*

Proof. From Equations 3.1, 3.4 and 3.5, we can derive

$$\begin{aligned} & Pr(SBF_N = 0) - Pr(SBF_{N-1} = 0) \\ =& Pr(SBF_N = 0 \mid A_{N-1}) Pr(A_{N-1}) \\ & + Pr(\bar{A}_N) - Pr(\bar{A}_{N-1}) \\ =& k(1-k)^{N-1}(Pr(SBF_N = 0 \mid A_{N-1}) - 1) \end{aligned} \tag{3.6}$$

Clearly, Eq. 3.6 exponentially converges to 0. i.e. $Pr(SBF_N[c]=0)$ converges at an exponential rate, and this is true for all cells in the SBF. Therefore, the expected fraction of 0s in the SBF converges at an exponential rate. ☐

Lemma 1 (Monotonicity). *The expected fraction of* 0*s in an SBF is monotonically non-increasing.*

Proof. Since the value of Eq. 3.6 is always no greater than 0, the probability that a cell becomes zero is always decreasing or remains the same. Combining the proof of Corollary 2, we can draw the conclusion. ☐

This lemma will be used to prove our general upper bound of the false positive rate where the number of iterations needs not to be infinity.

3.3.4 The Stable Point

Currently we know the fraction of 0s in an SBF will be a constant at some point, but we do not know the value of this constant. We call this constant the stable point.

Definition 2 (Stable Point). *The stable point is defined as the limit of the expected fraction of* 0*s in an SBF when the number of iterations goes to infinity. When this limit is reached, we call SBF stable.*

From Eq 3.1, we are unable to obtain the limit directly. However, we can derive it indirectly.

Theorem 2. *Given an SBF with m cells, if a cell is decremented by 1 with a constant probability p and set to Max with a constant probability k in each iteration, and if the probability that the cell becomes* 0 *at the end of iteration N is denoted by $Pr(SBF_N = 0)$,*

$$\lim_{N\to\infty} Pr(SBF_N = 0) = \left(\frac{1}{1 + \frac{1}{p(1/k-1)}}\right)^{Max} \tag{3.7}$$

Proof. The basic idea is to make use of the fact that SBF is stable, the expected fraction of 0,1,...,Max in SBF should be all constant.

Similar to the proof of Theorem 1, we can prove that

$$\lim_{N\to\infty} Pr(SBF_N = v)$$

exists, where $0 \leq v \leq Max$. In other words, when N is large enough, the probability that the cell becomes v is a constant. To simplify our notation, we will refer to $\lim_{N\to\infty} Pr(SBF_N = v)$ as $Pr(v)$ in this proof.

The key idea of this proof is as follows: since $Pr(v)$ is a constant, when the SBF is stable, the probability that the cell changes to v from some other values (e.g. $v + 1$) in each iteration must be the same as the probability that the cell changes from v to other values. Otherwise, $Pr(v)$ will either increase or decrease. Note that the probabilities of those changes caused by the decrementing and setting operations are all constants (p and k).

For example, we can consider the case of $Max = 3$ and $Pr(1)$. In each iteration, we decrement 1 from the cell with a constant probability p, and set the cell to Max with a constant probability k. Since $Pr(1)$ is a constant after the SBF becomes stable, the probability that the cell changes from 2 to 1 should be the same as the probability that the cell changes from 1 to 0 and Max. This can be expressed as

$$\begin{aligned} &Pr(2 \rightarrow 1 \mid SBF = 2) * Pr(2) \\ &\qquad = Pr(1 \rightarrow 0 \mid SBF = 1) * Pr(1) \\ &\qquad\qquad + Pr(1 \rightarrow Max \mid SBF = 1) * Pr'(1)) \end{aligned} \tag{3.8}$$

Note that $Pr'(1)$ is the probability of $SBF = 1$ after the decrementing but before the setting operation. Accordingly, we have

$$p * Pr(2) = p * Pr(1) + k * (p * Pr(2) - p * Pr(1) + Pr(1)).$$

The left hand side of the above equation is the probability of gaining 1s from the decrementing operation, and right hand side is the probability of losing 1s from both the decrementing and setting operations.

Using the same idea, the relationship among $Pr(0), Pr(1)$, ..., $Pr(Max)$ can be expressed as follows(when $Max \geq 3$) : when $v = 0$,

$$p * Pr(v+1) = k * (p * Pr(v+1) + Pr(v)), \tag{3.9}$$

when $0 < v < Max$,

$$p * Pr(v+1) = p * Pr(v) + k * (p * Pr(v+1) - p * Pr(v) + Pr(v)), \tag{3.10}$$

when $v = Max$,

$$k * (1 - Pr(v) + p * Pr(v)) = p * Pr(v). \tag{3.11}$$

From the above equations, we can derive

$$Pr(0) = (\frac{1}{1 + \frac{1}{p(1/k-1)}})^{Max}.$$

Currently this equation only holds when $Max >= 3$ since we have this assumption earlier. But it is not hard to verify that it holds when $Max = 1, 2$ as well using the same idea. □

The theorem can be verified by replacing the parameters in Eq. 3.1 with some testing values.

From Theorem 2 we know the probability that a cell becomes 0 when SBF is stable. If all cells have the same probability of being set, we can obtain the stable point easily. However, that requires the data stream to be uniformly distributed. Without this uniform distribution assumption, we have the following statement.

Theorem 3 (SBF Stable Point). *When an SBF is stable, the expected fraction of* 0s *in the SBF is no less than*

$$(\frac{1}{1 + \frac{1}{P(1/K-1/m)}})^{Max},$$

where K is the number of cells being set to Max and P is the number of cells decremented by 1 within each iteration.

Proof. The basic idea is to prove the case of $m = 2$ first, and generalize it to $m \geq 2$.

To simplify our notation, we refer to $\lim_{N\to\infty} Pr(SBF_N[c_i] = 0)$ as $Pr_i(0)$, $(i = 1, \ldots, m)$ and $(\frac{1}{1+\frac{1}{P(1/K-1/m)}})^{Max}$ as *LBound* in this proof.

We know that when all cells have the same probability of being set, the expected fraction of 0s in the SBF is *LBound*.

Now we prove that when cells have different probabilities of being set, the expected fraction of 0s in the SBF is no less than *LBound*. In other words, we need to prove when $k_1 = k_2 = \ldots = k_m = K/m$, the average probability

$$\frac{1}{m}\sum_{i=1}^{m} Pr_i(0) = \frac{1}{m}\sum_{i=1}^{m}(\frac{1}{1+ \text{-}1p(1/k_i - 1)})^{Max},$$

reaches the minimum value, where k_i is the probability that $SBF[c_i]$ is set. Note that $\sum_{i=1}^{m} k_i = K$.

To simplify the problem, we first prove when $m = 2$, $k_1 + k_2 = \alpha$,$(0 \leq \alpha \leq K)$, the above statement holds, i.e. when $k_1 = k_2 = \alpha/2$, $\frac{1}{1+\frac{1}{p(1/k_1-1)}} + \frac{1}{1+\frac{1}{p(1/k_2-1)}}$ is minimized. If $F(k_1)$ denote the above formula and $f(k_1)$ denote the first part, $F(k_1) = f(k_1) + f(\alpha - k_1)$. For $f(k_1)$, we can verify that the derivative of $f(k_1)$ w.r.t. k_1, $df/dk_1 < 0$, and $d^2f/dk_1^2 > 0$ when $k_1 \in [0, \alpha]$. Because of the symmetric property between $f(k_1)$ and $f(\alpha - k_1)$, the decreasing rate of $f(k_1)$ is faster than the increasing rate of $f(\alpha - k_1)$ when $k_1 \in [0, \alpha/2)$, thus $F(k_1)$ monotonically decreases within this period. When $k_1 = \alpha/2$, those two rates are equal. Symmetrically it increases monotonically when $k_1 \in (\alpha/2, \alpha]$. Therefore, when $k_1 = \alpha/2, F(k_1)$ is minimized.

Now we prove when $m > 2$, this conclusion also holds. First we assume $m = 2^j$, where j is an integer greater than zero. Later we will discuss the general case without this assumption. Since we have proved the case of two cells, we can group all cells into $m/2$ pairs arbitrarily. For each pair, we know if those two cells are set with an equal probability, the sum of $Pr_i(0)$ will decrease. For example, if we group $SBF[1]$ and $SBF[2]$ together, we can minimize $Pr_1(0) + PR_2(0)$ simply by changing both k_1 and k_2 into $(k_1 + k_2)/2$. This is the case for all other pairs. After the first grouping, we have at most $m/2$ different k_is. We then group those $m/2$ cells into pairs and minimized their sums again by balance those pairs. We can keep doing this until all cells are all balanced, i.e. all of them have the same probability being set. Now the sum is minimized because in each step the partial sum is minimized.

When the assumption $m = 2^j$ does not hold, we can divide one cell into two, where both have a same k_x and the sum does not change. For example, when $m = 3$, we can pick one cell arbitrarily, say $SBF[3]$, whose probability of being set is k_3. We can find a proper k_x, such that $f(k_3) = 2f(k_x)$. Therefore, we can transform all cases into what we have proved, i.e. $m = 2^j$. □

3.3.5 False Positive Rates

In our method, there could be two kinds of errors: false positives (FP) and false negatives (FN). A false positive happens when a distinct element is wrongly reported as duplicate; a false negative happens when a duplicate element is wrongly reported as distinct. We call their probabilities *false positive rates* and *false negative rates.*

Corollary 3 (FP Bound when Stable). *When an SBF is stable, the FP rate is a constant no greater than FPS,*

$$FPS = (1 - (\frac{1}{1 + \frac{1}{P(1/K-1/m)}})^{Max})^K. \tag{3.12}$$

Proof. If $Pr_j(0)$ denotes the probability that the cell $SBF[j] = 0$ when the SBF is stable, the FP rate is

$$\begin{aligned}&(\frac{1}{m}(1 - Pr_1(0)) + \cdots + \frac{1}{m}(1 - Pr_m(0))^K\\&= (1 - \frac{1}{m}(Pr_1(0) + \cdots + Pr_m(0)))^K\end{aligned}$$

Please note that $\frac{1}{m}(Pr_1(0) + \cdots + Pr_m(0))$ is the expected fraction of 0s in the SBF. According to Theorems 1 and 3, the FP rate is a constant and Eq. 3.12 is an upper bound of the FP rate. □

This upper bound can be reached when the stream elements are uniformly distributed.

Corollary 4 (The case of reaching the FP Bound). *Given an SBF with m cells, if the stream elements are uniformly distributed when the SBF is stable, the FP rate is FPS (Eq. 3.12).*

Proof. Because elements in the input data stream are uniformly distributed, each cell in the SBF will have the same probability to be set to Max. According to Theorem 1 and the proof of Theorem 3 we can derive this statement. ☐

Corollary 5 (General FP Bound). *Given an SBF with m cells, FPS (Eq. 3.12) is an upper bound for FP rates at all time points, i.e. before and after the SBF becomes stable.*

Proof. This can be easily derived from Lemma 1 and Corollary 3. ☐

Therefore, the upper bound for FP rates is valid no matter the SBF is stable or not.

From Eq. 3.12 we can see that m has little impact on FPS, since $1/m$ is negligible compared to $1/K$ ($m \gg K$). This means the amount of space has little impact on the FP bound once the other parameters are fixed. The value of P has a direct impact on FPS: the larger the value of P, the smaller the value of FPS. This can be seen intuitively: the faster the cells are cleared, the more 0s the SBF has, thus the smaller the value of FPS is. Oppositely, increasing the value of Max results in the increase of FPS. In contrast to P and Max, from the formula we can see the impact of the value of K on FPS is twofold: intuitively, using more hash functions increases the distinguishing power for duplicates (decreases FPS), but "fills" the SBF faster (increases FPS).

3.3.6 False Negative Rates

A false negative(FN) is an error when a duplicate element is wrongly reported as distinct. It is generated only by duplicate elements, and is related to the input data distribution, especially the distribution of gaps. A gap is the number of elements between a duplicate and its nearest predecessor.

Suppose a duplicate element x_i whose nearest predecessor is $x_{i-\delta_i}(x_i = x_{i-\delta_i})$ is hashed to K cells, $SBF[C_{i1}] \dots SBF[C_{iK}]$. An FN happens if any of those K cells is decremented to 0 during the δ_i iterations when x_i arrives. Let

$PR0(\delta_i, k_{ij})$ be the probability that cell C_{ij} ($j = 1 \dots K$) is decremented to 0 within the δ_i iterations. This probability can be computed as in Eq 3.1:

$$\begin{aligned} PR0(\delta_i, k_{ij}) = \sum_{l=Max}^{\delta_i - 1} &[Pr(SBF_{\delta_i} = 0 \mid A_l) Pr(A_l)] \\ &+ Pr(SBF_{\delta_i} = 0 \mid \bar{A}_{\delta_i}) Pr(\bar{A}_{\delta_i}), \end{aligned} \tag{3.13}$$

where

$$Pr(SBF_{\delta_i} = 0 \mid A_l) = \sum_{j=Max}^{l} \binom{l}{j} p^j (1-p)^{l-j}, \tag{3.14}$$

$$Pr(A_l) = (1 - k_{ij})^l k_{ij}, \tag{3.15}$$

$$Pr(SBF_{\delta_i} = 0 \mid \bar{A}_{\delta_i}) = \sum_{j=Max}^{\delta_i} \binom{\delta_i}{j} p^j (1-p)^{\delta_i - j}, \tag{3.16}$$

$$Pr(\bar{A}_{\delta_i}) = (1 - k_{ij})^{\delta_i}, \tag{3.17}$$

and k_{ij} is the probability that cell C_{ij} is set to Max in each iteration. The meanings of the other symbols are the same as those in the proof of Theorem 1. Also, most of above equations are similar, except that Eq. 3.16 is different from Eq. 3.4. This is because the initial value of the cell in the case of Theorem 1 is 0, but it is Max here.

Furthermore, the probability that an FN occurs when x_i arrives can be expressed as follows:

$$Pr(FN_i) = 1 - \prod_{j=1}^{K} (1 - PR0(\delta_i, k_{ij})). \tag{3.18}$$

When $\delta_i < Max$, $PR0(\delta_i, k_{ij})$ is 0, which means the FN rate is 0. Besides, for distinct elements who have no predecessors, the FN rates are 0. The value of δ_i depends on the input data stream. In the next section, we discuss how to adjust the parameters to minimize the FN rate under the condition that the FP rate is bounded within a user-specified threshold.

3.4 From Theory to Practice

In the previous section we proposed the SBF method and analytically studied some of its properties: stability, convergence rate, monotonicity, stable point, FP rates (upper bound) and FN rates. In this section, we discuss how SBF can be used in practice and how our analytical results can be applied.

3.4.1 Setting Parameters

Since FP rates can be bounded regardless of the input data but FN rates cannot, given a fixed amount of space, we can choose a combination of Max, K and P that minimizes the number of FNs under the condition that the FP rate is within a user-specified threshold. Meanwhile we take into account the time spent on each element, which is crucial in many data stream applications.

The expected number of FNs. Since our goal is to minimize the number of FNs, we can compute the expected number of FNs, $E(\#FN)$, as the sum of FN rates for each duplicate element in the stream: $E(\#FN) = \sum_{i=1}^{\widetilde{N}} Pr(FN_i)$, where $\widetilde{N}$ is the number of duplicates in the stream. Combining it with Eq. 3.18 we have

$$E(\#FN) = \sum_{i=1}^{\widetilde{N}} [1 - \prod_{j=1}^{K} (1 - PR0(\delta_i, k_{ij}))], \tag{3.19}$$

where δ_i is the number of elements between x_i and its predecessor, and k_{ij} is the probability that cell C_{ij} is set to Max in each iteration. C_{ij} is the cell element x_i is hashed to by the jth hash function. Since the function $PR0(\delta, k)$ is continuous, for each x_i there must be a $\bar{k}_i$ such that

$$(1 - PR0(\delta_i, \bar{k}_i))^K = \prod_{j=1}^{K} (1 - PR0(\delta_i, k_{ij})).$$

For the same reason, there must be an "average" $\widehat{\delta}$ and an "average" $\widehat{k}$ such that

$$\widetilde{N}[1-(1-PR0(\widehat{\delta},\widehat{k}))^K] = \sum_{i=1}^{\tilde{N}}[1-(1-PR0(\delta_i,\bar{k}_i))^K] = E(\#FN).$$

Let $f(\widehat{\delta},\widehat{k})$ be the average FN rate, i.e.

$$f(\widehat{\delta},\widehat{k}) = 1-(1-PR0(\widehat{\delta},\widehat{k}))^K. \tag{3.20}$$

Our task then becomes setting the parameters to minimize this average FN rate, $f(\widehat{\delta},\widehat{k})$, while bounding the FP rate within an acceptable threshold.

The setting of P. Suppose users specify a threshold FPS, indicating the acceptable FP rate. This threshold establishes a constraint between the parameters: Max, K, P, m and FPS according to Corollary 5. Thus, users can set P based on the other parameters:

$$P = \frac{1}{(\frac{1}{(1-FPS^{1/K})^{1/Max}}-1)(1/K-1/m)}. \tag{3.21}$$

Since m is usually much larger than K, $1/m$ is negligible in the above equation, which means that the setting of P is dominated only by FPS, Max, K, and is independent of the amount of space.

The setting of K. Since the FP constraint can be satisfied by properly choosing P , we can set K such that it minimizes the number of FNs. From the above discussions we know the relationship between the expected number of FNs and the probabilities that cells are set to Max. Next, we connect these probabilities with our parameters K, m and the input stream.

Suppose there are N elements in the stream of which n are distinct, and the frequency for each distinct element x_l is f_l'. Clearly $\sum_{l=1}^{n} f_l' = N$. Assuming that the hash functions are uniformly at random, for a cell that element x_i is hashed to, the number of times the cell is set to Max after seeing all N

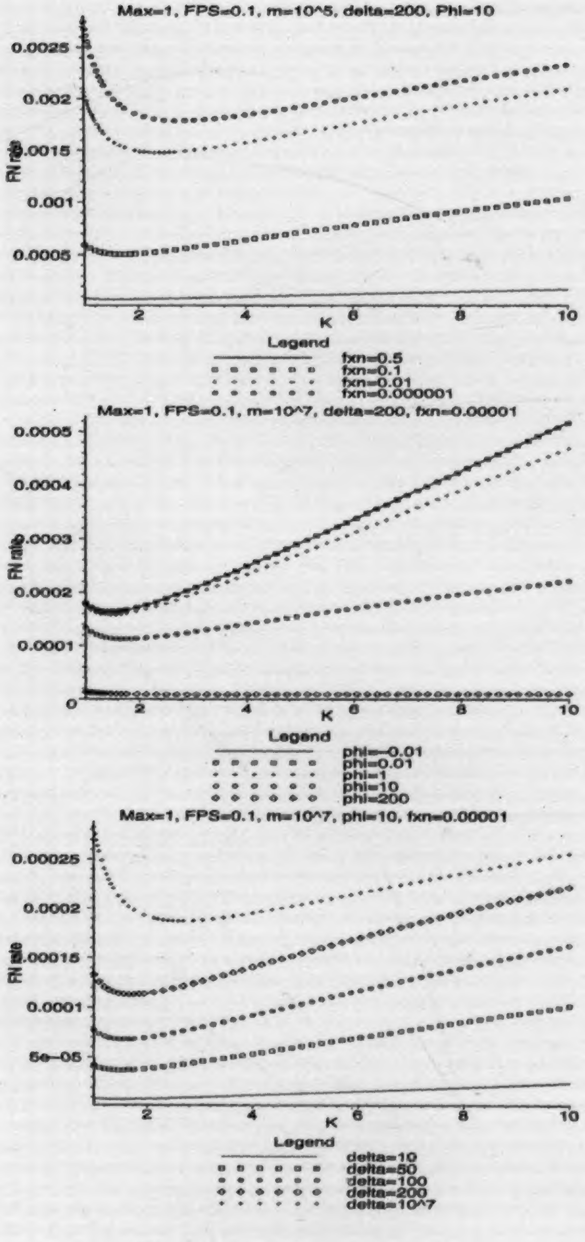


Figure 3.1: FN rates vs. K

elements is a random variable, $f_i + \sum_{l=1}^{n-1} f'_l I_l$, where f_i is the frequency of x_i in the stream, and each $I_l (l = 1 \ldots n-1)$ is an independent random variable following the Bernoulli distribution, i.e.

$$I_l = \begin{cases} 1, & Pr(I_l = 1) = \frac{K}{m}, \\ 0, & Pr(I_l = 0) = 1 - \frac{K}{m}. \end{cases}$$

Thus, $k_i = \frac{1}{N} f_i + \frac{1}{N} \sum_{l=1}^{n-1} f'_l I_l$ is also a random variable. For the K cells an element x_i is hashed to, the probabilities that those cells are set to Max in each iteration can be considered as K trials of k_i. Since the mean and the variance of each I_l are $\mu_{I_l} = \frac{K}{m}$ and $\sigma^2_{I_l} = \frac{K}{m}(1 - \frac{K}{m})$ respectively, it is not hard to derive that the mean and variance of k_i: $\mu_{k_i} = \frac{1}{N} f_i + \frac{1}{N}\frac{K}{m} \sum_{l=1}^{n-1} f'_l$ $= \frac{1}{N} f_i + \frac{K}{m}(1 - \frac{1}{N} f_i)$ and $\sigma^2_{k_i} = \frac{1}{N^2}\frac{K}{m}(1 - \frac{K}{m}) \sum_{l=1}^{n-1} f'^2_l$. Let

$$\phi_i = \frac{k_i - \mu_{k_i}}{\sqrt{\frac{K}{m}(1 - \frac{K}{m})}} = \frac{k_i - \frac{1}{N} f_i - \frac{K}{m}(1 - \frac{1}{N} f_i)}{\sqrt{\frac{K}{m}(1 - \frac{K}{m})}} \tag{3.22}$$

be a transformation on k_i. Then $\phi_i \in [-\frac{\frac{1}{N} f_i + \frac{K}{m}(1 - \frac{1}{N} f_i)}{\sqrt{\frac{K}{m}(1 - \frac{K}{m})}}, \frac{1 - \frac{1}{N} f_i - \frac{K}{m}(1 - \frac{1}{N} f_i)}{\sqrt{\frac{K}{m}(1 - \frac{K}{m})}}]$ is a random variable whose mean and variance are: $\mu_{\phi_i} = 0$ and $\sigma^2_{\phi_i} = \frac{1}{N^2} \sum_{l=1}^{n-1} f'^2_l$. Note that $\sigma^2_{\phi_i} \leq \frac{1}{N^2} \sum_{l=1}^{n-1} (f'_l f'_{max}) < \frac{1}{N^2} \sum_{l=1}^{n} (f'_l f'_{max}) = \frac{f'_{max}}{N}$, where f'_{max} is the frequency of the most frequent element in the stream. Since the mean and the variance of the random variable ϕ_i are independent of K and m, we may consider ϕ_i independent of K and m in practice. In other words, ϕ_i can be seen as a property of the input stream. Similar to $\bar{k}_i$ we can obtain a $\bar{\phi}_i$ such that

$$1 - (1 - PR0(\delta_i, \bar{\phi}_i))^K = 1 - \prod_{j=1}^{K} (1 - PR0(\delta_i, \phi_{ij})) = Pr(FN_i) \tag{3.23}$$

where $\bar{\phi}_i \in [Min(\phi_{ij}), Max(\phi_{ij})]$, and ϕ_{ij} are K trials of $\phi_i (j = 1 \ldots K)$. Since the standard deviation of ϕ_i is very small compared to the range of its possible values, and ϕ_i is considered independent of K and m, $\bar{\phi}_i$ can be approximately considered independent of K and m as well. For example, when $\frac{f'_{max}}{N} = 0.01$, $\frac{f_i}{N} = \frac{1}{10^6}$, $\frac{m}{K} = 10^6$, the value range of ϕ_i is approximately $[0, 1000]$, while $\sigma_{\phi_i} \leq 0.1$.

To set K, keeping all other parameters fixed we vary the values of K and compute the FN rate based on Equations 3.23, 3.13, 3.21 and 3.22. By trying different combinations of parameter settings ($Max = 1, 3, 7, 15$, $FPS = 0.2, 0.1, 0.01, 0.001$; $m = 1000, 10^5, 10^7, 10^9$; $\delta_i = 10, 100, 1000, 10^5, 10^7, 10^9$; $\frac{f_i}{N} = 0.5, 0.1; 0.01, 0.0001, 0.000001$ and $\bar{\phi}_i = 0.001, 0.1, 1, 10, 100, 1000, \ldots$), we find that once the values of FPS and Max are fixed, the value of the optimal or near optimal K is independent of the values of δ_i, f_i/N, $\bar{\phi}_i$ and m.

Observation 1. *The value of the optimal or near optimal K is dominated by Max and FPS. The input data and the amount of the space have little impact on it. Furthermore, the value is small (<10 in all of our testing).*

For example, when $FPS = 0.2$ and $Max = 1$, the value of the optimal or near optimal K is always between 1 and 2; when $FPS = 0.1$ and $Max = 3$, it is always between 2 and 3; when $FPS = 0.01$ and $Max = 3$, it is always between 4 and 5. Therefore, without considering the input data stream we can pre-compute the FN rates for different values of K based on Max and FPS and choose the optimal one. Our experimental results reported in the next section are consistent with this observation.

Figure 3.1 shows an example of how the FN rates change with different values of K under different parameter settings based in Eq. 3.23 and Eq. 3.13. From the figure we can see that in the case of $Max = 1$ and $FPS = 0.1$, we can set K to 2 regardless of the input stream and the amount of space. Therefore, in practice we can set ϕ_i, δ_i and f_i/N to some testing values (e.g. $0, 200, 0.00001$ respectively) and find the optimal or near optimal K using the formulas.

The setting of Max. Based on the above discussion, we can set K regardless of the input data, but to choose a proper value of Max, we need to consider the input. More specifically, to minimize the expected number of FNs, we need to know the distributions of gaps in the stream to try different possible values of Max on Eq. 3.20 and 3.13. Since the expected value of ϕ_i is 0 and its standard deviation is very small compared to its value domain, we set $\widehat{\phi}$ to 0 in the formulas.

To effectively use the space we only set Max to $2^d - 1$ (d is the number of bits/cell), otherwise the bits allocated for each cells are just wasted. Furthermore, in terms of the time cost, Max should be set as small as possible, because the larger Max is set, the larger P will be (see Eq.3.21, assuming K is a constant). For example, when $Max = 1$, $FPS = 0.01$ and $K = 3$(the optimal K), the computed value of P is 10; while $Max = 15$, $FPS = 0.01$, and $K = 6$ (the optimal K), the value of P computed is 141 (the value of P is not sensitive to m). In practice, we limit our choice of Max to 1, 3 and 7 (if higher time cost can be tolerated, larger values of Max can be tried similarly).

To choose a Max from these candidates, we try each value on Eq. 3.20 and Eq. 3.13, and find the one minimizing the average FN rate.

Figure 3.2 depicts the difference of average FN rates between $Max = 3$ and $Max = 1$ based on Eq.3.20 and Eq.3.13. We set $\frac{l_i}{N} = 0$ because we are considering the entire stream rather than a particular element in this case. The figure shows that if the values of gaps($\widehat{\delta}$) are smaller than a threshold, $Max = 3$ is a better choice. When the gaps become larger, $Max = 1$ is better. If the gaps are large enough, there is not much difference between the two options. The figure shows the cases under different settings of $\widehat{\phi}$, space sizes and acceptable FP rates.

We also tested the FN rate difference between $Max = 7$ and $Max = 3$, and observe the same general rule: a larger value of Max is better for smaller gaps, and a smaller FPS suggests a larger setting of Max. Similarly, we find no exceptions under other combinations of settings: $FPS = 0.2, 0.1, 0.01, 0.001$ and $m = 1000, 10^5, 10^7, 10^9$.

Trying different value of Max on Eq. 3.20 and Eq. 3.13, we set $\widehat{\phi}$ to 0 and assume that the distribution of the gaps are known. If the assumption cannot be satisfied in practice, we suggest setting Max to 1, because this setting

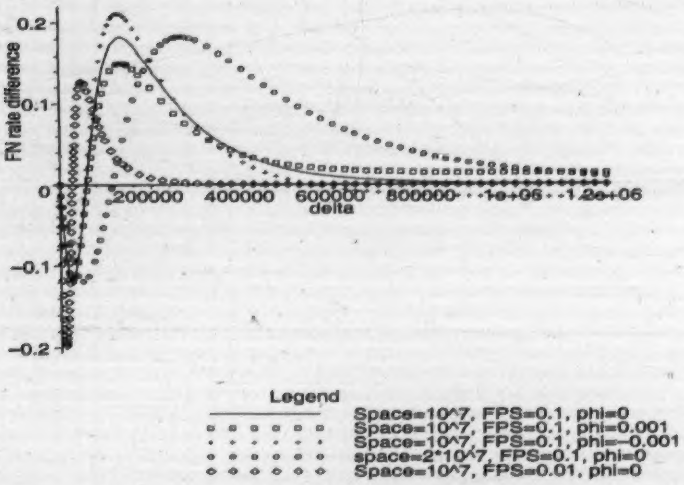


Figure 3.2: FN rates difference between $Max = 3$ and $Max = 1$ ($Max3 - Max1$) vs. gaps. K is set to the optimal value respectively under different settings.

often benefits a larger range of gaps in the stream. And our experiments also show that in most cases setting Max to 1 achieves better improvements in terms of error rates compared to the alternative method, LRU buffering. In fact, buffering performs well when gaps are small, which is similar to the cases that Max is larger. The behavior of our SBF becomes closer to the buffering method when the value of Max is set larger.

Summary of parameters setting and FP/FN tradeoff. In practice, given an FPS, the amount of available space and the gap distribution of the input data, to set the parameters properly, we first establish a constraint for P, which means P can be computed based on FPS, m, Max and K; then find the optimal values of K for each case of $Max(1, 3, 7)$ by trying limited number(≤ 10) of values of K on the FN rate formulas; Last, we estimate the expected number of FNs for each candidate value of Max using its corresponding optimal K and some prior knowledge of the stream, and thus choose the optimal value of Max. In the case that no prior knowledge of the input data is available, we suggest setting $Max = 1$. The described parameter setting process can be implemented within a few lines of codes. In addition, based on the above analysis, we obtain a constraint function between FN rates, FPS and other parameters, where the tradeoff between FPs and FNs can be clearly seen: when the other parameters are fixed, the larger the FP bound, the smaller the FN rate is.

3.4.2 Time Complexity

Since our goal is to minimize the error rates given a fixed amount of space and an acceptable FP rate, we do not discuss space complexity, and just focus on time complexity. There are several parameters to be set in our method: K, Max and P. Within each iteration, we firstly need to probe K cells to detect duplicates. After that we pick P cells and decrement 1 from them. Last we set the same K cells as probed in the first step to Max.

Therefore, the time cost of our algorithm for handling each element is dominated by K and P.

Theorem 4 (Time Complexity). *Given that K and Max are constants, processing each data stream element needs O(1) time, independent of the size of the space and the stream.*

Proof. From Eq.3.21 we know the constraint among K, P, m, Max and FPS(the user-specified upper bound of false positive rates). If K, Max and FPS are constants, the relationship between P and m is inversely proportional, which means m has no impact on the processing time. Since Max, K and FPS are all constants, the time complexity is O(1). ☐

Based on the discussion of parameter settings, we know that the selection of K is insensitive to m. Furthermore, the value of m and the stream size have little impact on the selection of Max based on our testing on Eq. 3.20. Therefore, our algorithm needs $O(1)$ time per element, independent of the size of space.

3.5 Experiments

In this section, we first describe our data set and the implementation details of 4 methods: SBF, Bloom Filter(BF), Buffering and FPBuffering (a variation of buffering which can be fairly compared to SBF). We then report some of the results on both real and synthetic data sets. Last, we summarize the comparison between different methods.

3.5.1 Data Sets

Real World Data. We simulated a web crawling scenario[24] as discussed in Section 3.1.1, using a Web crawl data set obtained from the Internet archive[8]. We hashed each URL in this collection to a 64-bit fingerprint using Rabin's method [107], as was done earlier [24]. With this fingerprinting technique, there is a very small chance that two different URLs are mapped to the same fingerprint. We verified the data set and did not find any collisions between the URLs. In the end, we obtained a 28GB data file that contained about 700 million fingerprints of links, representing a stream of URLs encountered in a Web crawling process.

Synthetic Data. We used 2 steps in generating the synthetic data. First, we generated a sequence of positive integers in which there was only one frequent integer appearing multiple times, and all others were distinct integers. Second, we scanned through the integer sequence generated in the first step, and randomly converted the only frequent integer into another integer with certain probability. We describe the details next.

In the first step, we used 2 different models to simulate the arrival positions of the only frequent integer in the sequence: the Poisson model and the b-model. The Poisson model is widely used in modeling real world traffic [43]; it is also shown that when the traffic has a bursty or a self-similar property, the b-model is more accurate [116].

The Poisson model had a parameter $\lambda(\leq 1)$, indicating the number of arrivals of the frequent integer within a unit time interval. Since we used a tuple-based stream model, there were no explicit timestamps for elements in our case. Instead, we used the element arrival positions as timestamps. As a result, when the arrivals of the only frequent integer followed a Poisson process with a parameter λ, the fraction of the frequent integer among an arbitrary number of elements is λ on average. It was also shown that the inter-arrival time (the number of elements in our case) between two consecutive arrivals followed an exponential distribution with a parameter $1/\lambda$ [43].

To simulate bursty traffics we used the b-model. The b-model had 3 parameters: bias $b(\geq 0.5)$, aggregation level l, and total volume v. When $b = 0.7$ and $l = 1$, the entire time interval under consideration was divided into 2 equal sub-intervals: $v * b$ traffic volume in the first sub-interval, and $v * (1 - b)$ in the second sub-interval. In other words, 70% of the total volume v was in the first sub-interval and 30% in the second sub-interval (or 30% in the first and 70% in the second, but we assumed the former case for the ease of presentation). If $l = 2$, we could further divide the first sub-interval into two equal finer sub-intervals, and allocate 70% of the volume in the higher level sub-interval into the first finer sub-interval and the remaining 30% into the second finer sub-interval. This process could be continued recursively for a larger l.

The traffic volume v in our case was the fraction of duplicates. As an example, $b = 0.7$ and $l = 1$ meant that 70% of the arrivals of the frequent integer were in the first half of the sequence, and 30% in the second half. However, it was possible that the predicted number of arrivals of the frequent integer in the first half was greater than the total number of arrivals in the first half. For example, if $v = 0.9$, the number of duplicates in the first half of the sequence was $0.9 * 0.7 = 0.63$, which meant that 63% of the total number of elements in the sequence should be allocated in the first half of the sequence. This was obviously infeasible. Observing this, we obtained a constraint between v, b and l: $l \leq ln(v)/ln(\frac{1}{2b})$. This constraint indicated that to obtain a larger aggregation levels l, the value of v and b should be smaller. For example, when $v = 0.1$ and $b = 0.7$, $l \leq 6$. But when $v = 0.5$ and $b = 0.8$, $l \leq 1$. When $b = 0.5$, there was no constraint because the recursive division process was always possible.

In the second step, we scanned through the integer sequence generated in the first step using the Poisson model or the b-model, and randomly converted the only frequent integer into another number i with a probability q_i, which followed an exponential distribution with a parameter u. After the second step, there were more than one frequent integer in the sequence, and the new arrival process of any frequent number i still followed the Poisson model or b-model, with a new parameter different from the old one by a factor of q_i (the Poisson case) or with the same parameter (the b-model case). By varying the parameters of the Poisson model or b-model in the first step, we were able to tune the arrival rate or the burstiness of duplicates. By varying the parameters of the exponential distribution in the second step, we were able to tune the fraction of distinct integers in the set of duplicates and the frequency distribution of those duplicates.

3.5.2 Implementation Issues

SBF Implementation. Our algorithm is simple and straightforward to implement: 1) hash each incoming stream element into K numbers using multiplication hash [44], and check the corresponding K cells; 2) generate a random number, decrement the corresponding cell and (P-1) cells adjacent to it by 1; 3) set those K cells checked in step 1 to Max. One issue we have to deal with is setting the parameters Max, K and P.

Based on the previous discussion, we can find some typical settings for different FPS regardless of different data sets. For example, for FPS=10%, we set Max=1, K=2 and P=4, which worked well for different data sets in our experiments. To evaluate our work, we implemented 3 alternative methods: Bloom Filters(BF), buffering and FPBuffering.

Bloom Filters Implementation. In our implementation, BF becomes a special case of SBF where Max=1 and P=0. Knowing the number of distinct elements and the amount of space, we can compute the optimal K (see the discussion in Section 2.3).

Buffering Implementation. Implementing buffering needs more work.

First, to detect duplicates we need to search the buffer. To speed up the searching process, we used a hash table, as was done by Broder et al. [24]. Second, when the buffer is full, we have to choose a policy to evict an old element and make room for the newly coming one. Broder et al. [24] compared 5 replacement policies for caching Web crawls. They showed that LRU and Clock, the latter of which is used as an approximation of LRU, were the best practical choices for the URL data set (there were some ideal but impractical ones as well); in terms of miss rate (FN rate in our case), there was almost no difference between these two though. We chose LRU in our experiments. Both LRU and clock need a separate data structure for buffering elements, so that we can choose one for eviction [24]. For simplicity of the implementation, we used a double linked list, while Broder et al. chose a heap. This difference should not affect our experimental results since our error rate comparison did not account for the extra space we used in buffering.

FPbuffering Implementation. To fairly and effectively compare our method to buffering method, we introduced a variation of buffering called FPbuffering. There are two reasons for this. First, SBF has both FPs and FNs while buffering has only FNs. In different applications the importance of FPs and FNs may be different. So it is hard to compare SBF to buffering directly. Second, the fraction of duplicates in the data stream is a dominant factor affecting the error rates, because FNs are only generated by duplicates and FPs by distincts. For buffering, a data stream full of duplicates will cause many FNs, while a stream consisting of all distincts cause no errors at all.

FPbuffering works as follows: when a new data stream element arrives, we search it in the buffer. If found, report duplicate as in the original buffering; if not found, we report it as a duplicate with a probability q, and as a distinct with probability $(1-q)$. In the original buffering, if an element is not found in the buffer, it is always reported as a distinct. This variation can increase the overall error rates of buffering when there are more distincts in the stream, but can decrease the error rates when there are more duplicates in the stream. Clearly, FPbuffering has both FPs and FNs. In fact, q is the FP rate since a distinct element will be reported as duplicate with a probability q. By setting a common FP rate with SBF, we can fairly compare their FN rates, and this comparison will not be affected by the fraction of duplicates in the stream.

In our experiments, we assumed that buffering and FPbuffering required 64 bits per URL fingerprint on the Web data (same as [24]) , and 32 bits per element on the synthetic data simulating the size of an IP address. In other words, each element occupies 64 bits for the real data and 32 bits for the synthetic data.

3.5.3 Theory Verification

In an experiment to verify some of our theoretical results, we tested the stable properties of our SBF and the convergence rate. The results are shown in Figures 3.3. From the graph we can see that the fraction of zeros in the SBF decreases until it becomes stable. When the allocated space is small, the convergence rate is higher. This is because when the space is larger, the probability a cell being set is smaller. From Corollary 2 and Eq 3.6 we know the convergence rate should be lower in this case. Also, we can see that when the SBF is stable, the fraction of zeros is still fluctuating slightly. This can be caused by the input data stream whose underlying distributions is varying. Furthermore, the fraction of 0s keeps decreasing in general before being stable; at this point, the FP rate should reach its maximum, and our theoretical upper bound for FP rates is also valid before the SBF become stable in this case. Our next experiments show the effectiveness of our theoretical FP bound. When the space is relatively small, the real FP rate is close to the bound.

3.5.4 Error Rates Comparison

This experiment compared the error rates between SBF, FPbuffering, buffering and BF on the real data by varying the size of the space. The real data set contained 694,984,445 URL fingerprints, of which 14.75% were distinct. To do the comparison under different fractions of distinct elements, we built two more real data sets by using the first 100,000 and 10 million elements of the

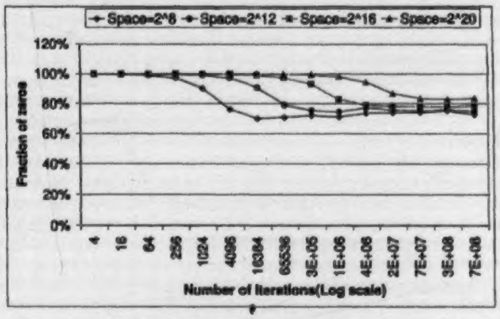


Figure 3.3: Fraction of zeros changed with time on the whole real data set (Max=1, K=2, P=4, FPS=10%), space unit=64bits

original data file. The fractions of distinct elements for these two data set respectively were 75.66% and 48.51%. For SBF, we set the acceptable FP rate (number of FPs/number of distincts), FPS, to 10%, and Max, K,P to 1, 2, 4 respectively. The results under different FPS settings will be shown in the next experiment. For FPbuffering, we set the FP rates to the same number as SBF so that both generated exactly the same number of FPs, and we can just compare their FN rates. Please note that buffering and BF only generate FNs and FPs respectively, and FPbuffering reduces the FN rates of buffering substantially in most cases by introducing a certain amount of FPs.

Comparison between different methods. The tables in Figure 3.4 show that when the space is relatively small, SBF is better. SBF beats FPbuffering by 3-13% in terms of FN rate on different data sets, when their FP rates are the same. For the problem we are studying, we think this amount of improvement is nontrivial for 2 reasons. First, Broder et al. [24] implemented a theoretically optimal buffering algorithm called MIN, where they assume "the entire sequence of requests is known in advance", and accordingly choose the best replacement strategy. Even this obviously impractical and ideal algorithm can only reduce the miss rates (FN rates in our case) of the LRU buffering, by no more than 5% in about 2/3 region (different buffer sizes). Second, from the tables we can see that even increasing the amount of the space by a factor of 4, the FN rates for buffering can be decreased by around 10-20%, which means the improvement from SBF may be equivalent of that

76% Distinct(100K elements)

Space (bits)	FPBuffering FN Rate	SBF FN Rate	SBF&FPBuffering FP Rate	Buffering FN Rate	BF FP Rate
16384	46%	35%	8.4%	50%	83.8%
65536	35%	23%	6.7%	37%	43.4%
262144	24%	11%	3.0%	25%	7.2%
1048576	9%	4%	0.4%	9%	0.6%
4194304	0.1%	1%	0.1%	0.1%	0.1%

49% Distinct (10M elements)

Space (bits)	FPBuffering FN Rate	SBF FN Rate	SBF&FPBuffering FP Rate	Buffering FN Rate	BF FP Rate
16384	60%	54%	8.1%	65.7%	99.7%
262144	48%	40%	6.6%	51.5%	95.9%
4194304	30%	23%	4.5%	31.7%	43.5%
67108864	11%	5%	0.5%	11.0%	0.7%
1073741824	0.0%	0.4%	0.1% (only SBF)	0.0%	0.1%

15% Distinct (695M elements)

Space (bits)	FPBuffering FN Rate	SBF FN Rate	SBF&FPBuffering FP Rate	Buffering FN Rate	BF FP Rate
16384	71%	68%	8.2%	78%	99.99%
262144	65%	60%	7.0%	70%	99.81%
4194304	55%	50%	5.7%	58%	96.93%
67108864	43%	36%	3.5%	45%	54.08%
1073741824	17%	13%	1.6%	17%	2.65%
4294967296	2%	5%	1.6%	2%	1.87%

Figure 3.4: Error rates comparison between SBF, FPBuffering, Buffering and BF

from doubling the amount of space. The FP rates of BF is much higher than the acceptable FP rates in the first 2-3 rows of each table. Since buffering only generates FNs, it is not comparable to SBF here. But we can see that the FN rates of FPbuffering also decrease by introducing FPs into it.

However, we also notice that when the space is relatively large (the last row of each table), SBF performs not as good as buffering and BF. This is because when the space is large, BF might be able to hold all the distincts and keep a reasonable FP rates. We can directly compute the amount of space required based on the FP rates desired and the number of distincts in the data set according to the formula in Section 2.3. In this case, there is no need to evict elements out of the BF, which means SBF is not applicable. If we can afford even more space, which is large enough to hold all the distincts in the data set using a buffer, there will be no errors at all. The last row of the second table shows this scenario. But in many data stream applications, a fast storage is needed to satisfy real time constraints and the size of this storage is typically less than the universe of the stream elements as discussed in Introduction.

Another fact is that in both SBF and buffering, we can refresh the storage and bias it towards recent data; they both evict stale elements continuously and keep those fresh ones. While BF is not applicable in this case since BF can be only used to represent a static data set. Thus, it is not useful in many data stream scenarios that require dynamic updates.

Varying acceptable FP rates. Another experiment we ran was to test the effect of changing the acceptable FP rates. The results are shown in Figure 3.5. In this experiments, we set Max=3, K=4 when acceptable FP rates are set to 0.5% and 1%, and set Max=1, K=2 when acceptable FP rates are set to 10% and 20%. The bar chart depicts the FN rate difference between FPbuffering and SBF. Again, the FP rates of both methods are set to the same number. Clearly, it shows that the more FPs are allow, the better SBF performs.

Comparison on different data sets. To see the impact of different data sets on both the SBF and FPBuffering methods, we ran similar experiments on the synthetic data. In general, the results were consistent with those from

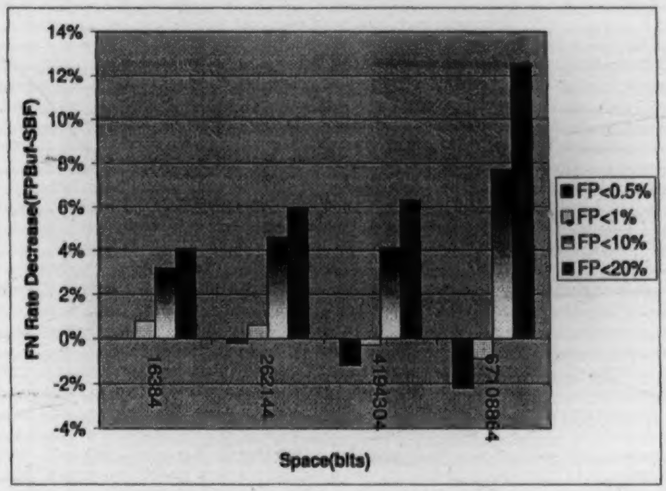


Figure 3.5: FN rate differences between FPBuffering and SBF varying allowable FP rate(695M elements)

the real data.

The acceptable FP rate was set to 10%, and the parameters for SBF were as follows: $Max = 1$, $K = 2$ and $P = 4$. In these data sets, the stream sizes were all fixed to 2^{24} (about 16 million elements). The parameters used to generate the data sets are shown in Figure 3.6.

Between the two data sets shown in Figure 3.6(a) and 3.6(b), the only difference is the bias parameter b, which indicates the burstiness of the data sets. These two sub-figures show that the FN rate difference between SBF and FPBuffering becomes smaller when the data set has the bursty property ($b = 0.7$) compared to the case that the data set is not bursty ($b = 0.5$). But the impact of this burstiness is not significant.

We also set b to 0.7 and change the exponential distribution parameter u, which controls the frequency distribution of duplicates, from 10^3 to 10^6. In the data set generated under this setting, the gaps between duplicates are increased in general, because increasing u means that the exponential distribution becomes less skewed, and the number of distinct elements in the set of frequent elements is increased (we consider the first time that a frequent element appears as a distinct element), and thus the probability of generating two identical integers within a short period becomes smaller. Figure 3.6(c) shows the results obtained from experiments run on this data set. It demonstrates that both SBF and FPBuffering need more space to reach a particular FN rate when the gaps are increased. In the region with the FN rate 90%, the space allocated to both methods is too small and does not help much, because without any space cost one can obtain similar results by randomly predicting each element as a duplicate at a probability 0.1.

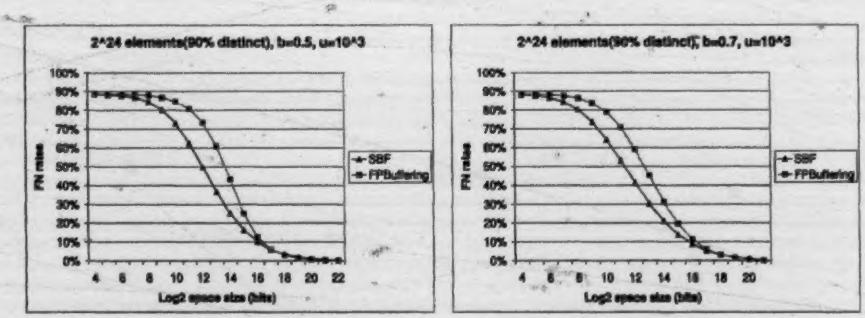
Figure 3.6(d) depicts the results from the experiments run on the Poisson data set. u is set to the same value as in the previous sub-figure. Since there are more duplicates in this data set, the gaps in general are smaller compared to the previous data set. As a result, to reach a certain FN rate, both methods need less space. From this and some others experiments whose results are not shown here we find that SBF consistently outperforms FPBuffering on datasets generated using both the Poisson and the b- models.

Besides, all sub-figures in Figure 3.6 also show that SBF can beat FPBuffering up to 30% in terms of FN rates when both methods have exactly the same FP rate ($\leq$ 10%). When the allocated space is large (but not large enough to store all distinct elements), FPBuffering may be more accurate than SBF. But in this case, as discussed in the comparison on the real world data, the original BF method (which is also a special case of SBF) is usually the best since it only generates FPs under the acceptable threshold and no FNs. Of course, if the available space is large enough to store all distinct elements, buffering is the best in terms of accuracy, since it is precise.

3.5.5 Time Comparison

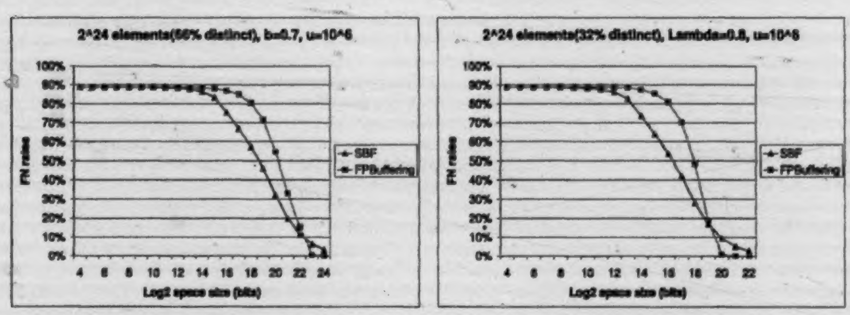
As discussed in the implementation section, SBF and BF need O(1) time to process each element. The exact time depends on the the parameter settings. For example, when K=2 and P=4, SBF needs less than 10 operations within each iteration.

For buffering and FPbuffering, their processing time is the same. It depends on 2 processes: element searching and element evicting. Searching can



(a) B-model dataset 1 (aggregation level $l = 6$).

(b) B-model dataset 2 (aggregation level $l = 6$).



(c) B-model dataset 3 (aggregation level $l = 6$).

(d) Poisson dataset

Figure 3.6: FN rate comparisons between SBF and FPBuffering on synthetic data sets (FP rates $\leq 10\%$, in the SBF method $Max = 1$, $K = 2$ and $P = 4$).

be quite expensive without an index structure. Both our experiments and those of Broder et al.[24] used a hash table to accelerate the search process. The extra space that is needed for a hash table to keep the search time constant is linear in the number of elements stored. The process of maintaining the LRU replacement policy(finding the least recently used element) is also costly, and extra space is needed to make it faster. This extra space can be quite large for LRU. However, this cost can be reduced to 2 bits per elements by using the Clock approximation of LRU [24].

Therefore, buffering and FPbuffering need extra space linear in the number of buffer entries to reach a similar O(1) processing time. But in our error rate comparison, we did not count this extra space for buffering and FPbuffering.

3.5.6 Methods Comparison Summary

We compared 4 methods in this section: SBF, BF, FPbuffering and buffering. Among them, BF and buffering have only FPs and FNs respectively, and SBF and FPbuffering have errors of both sides.

BF is a space efficient data structure which has been studied in the past and is widely used. It is good for representing a static set of data provided that the number of distinct elements is known. However, in data stream environments, the data is not static and it keeps changing. Usually it is hard to know the number of distinct elements in advance. Moreover, BF is not applicable in cases where dynamic updates are needed since elements can only be inserted into BF, but cannot be dropped out. Consequently, BF is not suitable for many data stream applications.

SBF, buffering and FPBuffering can be all applied to data stream scenarios. SBF is better in terms of accuracy and time when certain amount of FP rates are acceptable and the space is relatively small, which is the case in many data stream applications due to the real-time constraint. When the space is relatively large or only small FP rates are allowed, buffering is better.

3.6 Related Work

The recent work of Metwally et al.[96] also study the duplicate detection problem in a streaming environment based on Bloom filters(BF) [17]. They consider different window models: Landmark windows, sliding windows and jumping windows. For the landmark window model, which is the scenario we consider, they apply the original Bloom filters without variations to detect duplicates, and thus do not consider the case that the BFs become "full". For the sliding window model, they use counting BFs [66] (change bits into counters) to allow removing old information out of the Bloom filter. However, this can be done only when the element to be removed is known, which is not possible in many streaming cases. For example, if the oldest element needs to be removed, one has to know that which counters are touched by the oldest element, but this information cannot be found in counting BFs, and maintaining this knowledge can be quite expensive. For the jumping window model, they cut a large jumping window into multiple sub-windows, and represent both the jumping window and the sub-windows with counting BFs of the same size. Thus, the jumping window can "jump" forward by adding and removing sub-window BFs.

Another solution for detecting duplicates in a streaming environment is the buffering or the caching method, which has been studied in many areas such as database systems [69], computer architecture [76], operating systems [112], and more recently URL caching in Web crawling [24]. We compare our method with those of Broder et al.[24] in the experiments.

The problem of exact duplicate elimination is well studied, and there are many efficient algorithms(e.g. see [69] for details and references). For the problem of approximate membership testing in a non-streaming environment, the Bloom filter has been frequently used (e.g. [92, 91]) and occasionally extended [66, 98]. Cohen and Matias[42] extend the Bloom filter to answer multiplicity queries. Counting distinct elements using the Bloom filter is proposed by Whang et al.[119]. Another branch of duplicate detection focus on fuzzy duplicates [35, 5, 16, 118], where the distinction between elements is not straightforward to see.

A related problem to duplicate detection is counting the number of distinct elements. Flajolet and Martin[67] propose a bitmap sketch to address this problem in a large data set. The same problem is also studied by Cormode et al. [45], and a sketch based on stable random variables is introduced. Besides, the sticky sampling algorithm of Manku and Motwani [94] also randomly increment and decrement counters storing the frequencies of stream elements, but the decrement frequency is varying and not for each incoming element. Their goal is to find the frequent items in a data stream.

As for data stream systems [9, 27, 114, 28, 51, 37], as far as we know, most of them divide the potentially unbounded data stream into windows with limited size and solve the problem precisely within the window. For example, Tucker et al. introduce punctuations into data streams, and thus duplicate eliminations could be implemented within data stream windows using traditional methods[114].

Since there is no way to store the entire history of an infinite data stream using limited space, our SBF essentially represents the most recent information by discarding those stale information continuously. This is useful in many scenarios where the recent data is more important and this importance decays over time. A number of such kinds of applications are provided in [41] and [104]. Our motivating example of web crawling also has this property, since it may not matter that much to redundantly fetch a Web page that have been crawled a long time ago compared to fetching a page that have been crawled more recently.

3.7 Summary and Possible Extension

In this chapter, we propose the SBF method to approximately detect duplicates for streaming data. SBF extends the Bloom filter to allow outdated information to be removed from the summary structure so as to have enough space to represent fresh data. We formally and experimentally study the properties of our algorithm including stability, exponential convergence rates, monotonicity, stable points, bounded FP rates, FN rates dependent on input data stream, O(1) time independent of the stream size and space.

We empirically evaluate SBF and report the conditions under which our method outperforms the alternative methods, in particular buffering and FP-Buffering methods using LRU replacement policy, both of which need extra space linear in the buffer size to obtain constant or nearly constant processing time.

Our SBF method works in a "landmark" window environment where the starting point of the window is fixed. Since an element arrives recently has less chance being "kicked out", SBF implicitly favors recent data. But no explicit sliding windows can be specified by users to indicate the "recentness" range. We are working on the sliding version now.

Chapter 4

Approximate Frequency Query Processing

In this chapter, after introducing frequency queries and multipurpose sketching techniques, we first propose a new unbiased estimation algorithm, referred to as CMM, based on Count-min sketches [48] to approximately answer multiplicity queries of data streams. Our experiments on both synthetic and real data sets show that the new algorithm gives much more accurate results (e.g. orders of magnitude improvement on the real data set) than the CM estimation algorithm on a wide range of data sets except when data is highly skewed.

Second, we show through theoretical analyses and experimental evaluations that CMM performs very similarly to the algorithms based on Fast-AGMS sketches [31, 46], and all the analytical results reported for Fast-AGMS [31] also hold for Count-min with our CMM algorithm. Hence, Count-min can be also applied to the cases where Fast-AGMS is used as a building block without losing accuracy, time and space efficiency (e.g. [81] and [68]).

Having two estimation options with different properties, Count-min can do more than Fast-AGMS. For example, the CM estimation approach provides one-sided error approximations, which can be very useful in some cases. In finding frequent elements in a data stream, all candidates whose multiplicities exceed a given threshold are guaranteed to be returned using the CM estimation. Multiplicity estimates for all qualified candidates can be obtained using our CMM approach since it is usually much more accurate in practice. In contrast, Fast-AGMS fails to provide this deterministic guarantee no matter how much space is given. In addition, Count-min with the CM estimation is more accurate than Fast-AGMS when the data set is highly skewed, and CM has a better space bound, meaning that given an error bound and a confidence interval, Count-min using CM needs less space than Fast-AGMS.

Third, we propose a new unbiased algorithm for self-join size estimations based on Count-min sketches. Unless there is a confusion, we will also refer to this algorithm as CMM. Similarly, the accuracy of this algorithm is much better than the previous Count-min estimation algorithm (also referred to as CM) in practice on a wide range of data sets except when the data set is highly skewed. Through our analytical and empirical evaluations we show that CMM performs very similarly to Fast-AGMS in terms of self-join size estimations. Again, having two estimation approaches with different properties makes Count-min a more powerful and flexible data stream summary.

The rest of this chapter is organized as follows. Section 4.1 describes the queries to be answered and the importance of those queries. Then we introduce our CMM estimation algorithm for multiplicity queries in Section 4.2, where analyses and experimental results are also provided. Section 4.3 discusses our CMM algorithm for self-join size estimations. In Section 4.4 we describe the research work closely related to CMM. Section 4.5 summarize this chapter.

4.1 Frequency Queries and Multipurpose Sketching.

As discussed in the previous chapters, different time and space efficient sketching techniques have been proposed, some dedicated to one type of query, and a few others such as Count-min [48] and Fast-AGMS sketch (a.k.a Count-sketch) [31] can be used to answer multiple queries. Two important queries that can be answered using Count-min or Fast-AGMS sketches are multiplicity queries and self-join size estimations. Although our estimation algorithms can be extended to answering other queries, in this chapter we focus on these two queries.

A *multiplicity query*, also called a point query or a frequency query, is to find the number of times a given element appears in a data stream. This is an important query because the techniques for answering multiplicity queries can be often applied to answer other frequency related queries such as iceberg queries [42] (where the goal is to find the elements whose frequencies exceed a threshold), finding top-K frequent elements [31], range queries [48] (where the goal is to find the sum of frequencies of elements within a range), and approximating quantiles [48]. Multiplicity queries are also important in traditional non-streaming settings (see [42] for more examples).

The *self-join size*, also known as the second frequency moment, of a multiset is $\sum_{i\in D} f_i^2$, where D is the domain from which the values are drawn, and f_i is the frequency of value i. The self-join size indicates the degree of skew of a data set. For data distributions such as Zipfian and exponential, the self-join size uniquely determines the parameter of the distribution [3]. Knowing the parameter of a distribution can improve the accuracy of estimations. For instance, in answering multiplicity queries, we can compute the Zipfian parameter of the input data stream (assuming it follows the Zipfian model), and accordingly choose an algorithm between the new one we propose in this chapter and the one previously proposed since both algorithms are based on the same sketch. As another example, the self-join size can be also useful in selecting an optimal sampling strategy to estimate the number of distinct values [75]. More applications of the self-join size can be found in [4] and [3].

To answer multiplicity queries and self-join size estimations, we focus on Count-min sketches [48], which have been implemented on an operational data stream monitoring systems, AT&T's Gigascope [51, 47], for real-time IP traffic analyses (including multiplicity queries and self-join size estimations) and for other operational reasons [85]. Count-min has some nice properties such as one-sided errors and better space bounds (smaller by a factor of $1/\epsilon$, where ϵ is the relative error) in comparison with the best known alternative sketching techniques. However, better space bounds may not always guarantee better performance in practice. Based on our experiments, we find that the previous estimation algorithms using Count-min, referred to as CM, are not as accurate as those using Fast-AGMS [31] on a wide range of data sets. On slightly skewed or uniformly distributed data sets, in particular, Fast-AGMS performs significantly better. In this chapter, we demonstrate that Count-min sketches can actually do as well as Fast-AGMS both in theory and in practice regardless of the data distributions using our new estimation algorithms. Furthermore, our new estimation algorithms can be combined with those previous algorithms without conflicts, hence making Count-min a more powerful and flexible sketch.

Next, we introduce our new estimation algorithms for multiplicity queries based on Count-min sketches.

4.2 Unbiased Estimates for Multiplicity Queries using Count-min Sketches

The estimation procedures described in Section 2.1.1 give upper bounds of the true values. We propose our estimation methods, *count-mean-min (CMM)* , which gives unbiased estimates for both multiplicity queries and self-join size estimations using exactly the same count-min sketch. We discuss the multiplicity query case in this section.

4.2.1 Basic Idea

Recall the estimation procedure of CM: given a query element q and hash functions h_i $(i = 0, \dots, d-1)$, the frequency estimate $\hat{f}_q$ is the minimum value of the counters q has touched (i.e. $CM[i, h_i(q)]$, $i = 0, \ldots, d-1$). Usually the counters q touches are also touched by other elements, thus even the minimal counter value is expected to be larger than the true value f_q. The source of the error is the contributions of other elements to the counters $CM[i, h_i(q)]$. We characterize the contributions made by elements other than q to the counters $CM[i, h_i(q)]$ as *noise*. The CM algorithm returns the counter value with the least noise. Our CMM algorithm tries to estimate the noise in each counter, removes the noise and returns the residue.

Of course we do not know exactly the value of the noise since the noise is a random variable, but we can estimate its expected value. For a counter $CM[i, h_i(q)]$, the noise can be estimated from the values of all other counters not touched by q in that row i. The value of each counter not touched by q can be considered as an independent random variable following the same distribution as the noise, assuming that the hash functions map each element i to the range $[0, d-1]$ uniformly at random (pair-wise independence [99] is sufficiently for our theoretical results in this section). In fact, for a multiplicity query, the values of the counters that are not touched by the query element q in row i demonstrate the probability distribution of the noise in counter $CM[i, h_i(q)]$.

4.2.2 Our Estimation Algorithm

Given a query element q, we use the same set of hash functions h_i ($i = 0, \ldots, d-1$) as used in constructing the Count-min sketch, and check the d counters q is mapped to, i.e. $CM[i, h_i(q)]$ ($i = 0, \ldots, d-1$). Instead of returning the minimum value of the d counters, we deduct the value of estimated noise from each of those d counters, and return the median of the d residues. The estimated noise in each counter $CM[i, h_i(q)]$ can be computed as the average value of all counters in row i except counter $CM[i, h_i(q)]$ itself. That is, the noise is estimated to be $(N - CM[i, h_i(q)])/(w-1)$, where N is the stream size and w is the sketch width.

4.2.3 Analyses of Our Algorithm

Since for each row of the sketch, the analysis is the same, we just discuss the case for a particular row i. Let X_x be a Bernoulli random variable indicating if element x is hashed to the same counter that the query element q is hashed to, i.e.

$$X_x = \begin{cases} 1, & x \text{ is hashed to the same counter as } q \text{ is;} \\ 0, & \text{otherwise.} \end{cases}$$

Assuming that the hash function maps each element to one of the w counters uniformly at random, the probabilities of the above two cases are as follows: $Pr[X_x = 1] = 1/w$ and $Pr[X_x = 0] = 1 - 1/w$. The value of the counter q is hashed to is also a random variable, $f_q + \sum_{x \neq q} f_x X_x$, where f_q and f_x are the true frequencies of q and x respectively.

Lemma 2. *Given a hash function picked uniformly at random from a pairwise independent family, for a multiplicity query of element q and each row of the sketch, our CMM estimate $\hat{f}_q$ is expected to be f_q, and the variance is $\frac{1}{w-1}\sum_{x \neq q} f_x^2$, where w is the sketch width.*

Proof. We only discuss the case for row i of the sketch here since the analysis for other rows is exactly the same. Recall that our estimation procedure described in Section 4.2.2 is to deduct an estimated noise, the average value of other counters, from the counter q is hashed to. Thus, the estimate

$$\begin{aligned} \hat{f}_q &= CM[i, h_i(q)] - \frac{1}{w-1}(N - CM[i, h_i(q)]) \\ &= f_q + \sum_{x \neq q} f_x X_x - \frac{1}{w-1}(N - (f_q + \sum_{x \neq q} f_x X_x)) \\ &= f_q + \frac{w}{w-1} \sum_{x \neq q} f_x X_x - \frac{1}{w-1}(N - f_q) \\ &= f_q + \frac{1}{w-1} \sum_{x \neq q} w f_x X_x - \frac{1}{w-1} \sum_{x \neq q} f_x \\ &= f_q + \frac{1}{w-1} \sum_{x \neq q} f_x (w X_x - 1). \end{aligned}$$

The expectation of $\hat{f}_q$

$$\begin{aligned} E[\hat{f}_q] &= f_q + \frac{1}{w-1}\sum_{x\neq q} f_x(wE[X_x]-1) \\ &= f_q + \frac{1}{w-1}\sum_{x\neq q} f_x(w\frac{1}{w}-1) = f_q. \end{aligned}$$

Therefore, $\hat{f}_q$ is an unbiased estimate.

The variance of $\hat{f}_q$

$$\begin{aligned} VAR[\hat{f}_q] &= E[(\hat{f}_q - E[\hat{f}_q])^2] \\ &= E[(\frac{1}{w-1}\sum_{x\neq q} f_x(wX_x-1))^2]. \end{aligned}$$

To simplify the formula, let $Z_x = wX_x - 1$, then

$$E[Z_x] = 0$$

$$E[Z_x^2] = E[w^2X_x^2 - 2wX_i + 1] = w - 1.$$

Thus,

$$VAR[\hat{f}_q] = \frac{1}{(w-1)^2}E[(\sum_{x\neq q} f_xZ_x)^2]$$

$$\begin{aligned} &= \frac{1}{(w-1)^2}E[\sum_{x\neq q} f_x^2Z_x^2 + 2\sum_{x\neq q}\sum_{\substack{y\neq q\\ y<x}} f_xf_yZ_xZ_y] \\ &\quad (\because Z_x, Z_y \text{ pairwise independent}) \\ &= \frac{1}{(w-1)^2}(\sum_{x\neq q} f_x^2E[Z_x^2] + 2\sum_{x\neq q}\sum_{\substack{y\neq q\\ y<x}} f_xf_yE[Z_x]E[Z_y]) \\ &= \frac{1}{w-1}\sum_{x\neq q} f_x^2. \end{aligned}$$

□

Comparison with Fast-AGMS. Fast-AGMS [31] can be used to answer multiplicity queries as well. It is not hard to show the following statement.

Lemma 3. *Given sketches of the same width and depth and using independent hash functions, the expectation and variance of the estimates from Fast-AGMS are the same as those from our CMM algorithm.*

Proof. Similar to the proof of Lemma 2, we can obtain the expected value and the variance of the Fast-AGMS estimate; the expectation is the same as that of CMM's, and the variance is $\frac{1}{w}\sum_{x \neq q} f_x^2$. For a given element q, let $\bar{f}_q$ be the frequency estimate from a Fast-AGMS sketch with width w and depth d. X_x is still defined as the same indicator variable, indicating whether element x is hashed to the same counter as q is. Let Y_x be another random variable indicating x is hashed to either -1 or 1 with the same probability $1/2$ by another independent hash function. That is

$$Y_x = \begin{cases} 1, & Pr[Y_x = 1] = \frac{1}{2} \\ -1, & Pr[Y_x = -1] = \frac{1}{2}. \end{cases}$$

then we have

$$\begin{aligned}
&\bar{f}_q = f_q + \sum_{x\neq q} f_x X_x Y_x, \\
&E[\bar{f}_q] = f_q + \sum_{x\neq q} f_x E[X_x]E[Y_x] = f_q, \\
&VAR[\bar{f}_q] = E[(\bar{f}_q - f_q)^2] = E[(\sum_{x\neq q} f_x X_x Y_x)^2] \\
= \;& E[\sum_{x\neq q} f_x^2 X_x^2 Y_x^2] + E[2\sum_{x\neq q}\sum_{\substack{y\neq q\\ y<x}} f_x f_y X_x X_y Y_x Y_y] \\
&(\because X_x X_y, Y_x Y_y \text{pairwise independent}) \\
= \;& \sum_{x\neq q} f_x^2 E[X_x^2]E[Y_x^2] + \\
& 2\sum_{x\neq q}\sum_{\substack{y\neq q\\ y<x}} f_x f_y E[X_x]E[X_y]E[Y_x]E[Y_y] \\
= \;& \frac{1}{w}\sum_{x\neq q} f_x^2.
\end{aligned}$$

Recall that the variance of our CMM estimate is $\frac{1}{w-1}\sum_{x\neq q} f_x^2$, meaning that if CMM is given one more counter in each row, the variances of these two methods will be exactly the same. Given that CMM needs one less hash function in each row, and this can lead to some saving in the storage of the hash functions, we consider the two variances the same. Even if there is any, the difference is negligible especially when the depth of the sketch is small due to the time cost. □

Theorem 5. *The analytical results reported for Fast-AGMS [31] are all applicable to the Count-min sketch using the CMM algorithm.*

Proof. Because the expectations and variances of the two methods are the same, all proofs in [31] can be adapted to our CMM estimation. See [31] for the detailed proof. Note that the presentation style and some of the notations in [31] are different from ours. □

4.2.4 Experiments for Multiplicity Queries

In this section, we experimentally compare CMM to the related estimation algorithms: CM, Fast-AGMS and SBF with the MI heuristic (see the Spectral Bloom filters part in Section 2.1.1).

Implementation issues. In our CMM algorithm for answering multiplicity queries, we also use the median of all counters in a sketch row as the estimated noise besides using the mean as described in the algorithm, because median is less sensitive to outliers in data values. Computing the median of the counters not touched by the query element for each query is costly. To improve the time efficiency, we consider the median of all counters as the noise, which can be obtained once and used for all queries. This estimate is still accurate because the median of all counters in one sketch row is approximately the same as the median of that row with one less counter.

To further increase accuracy for both CMM and Fast-AGMS, we return 0 if CMM or Fast-AGMS gives a negative estimate since the estimate is clearly wrong. Similarly, if CMM gives an estimate larger than the one from CM, we return the latter instead since an estimate above the upper bound is also obviously wrong. Having multiple estimates from multiple sketch rows, we return the median as the final estimate for both CMM and Fast-AGMS. The hash functions we use are obtained from MassDal [95].

Synthetic and real data sets. We generated synthetic data sets whose element frequencies followed Zipfian distributions with different Zipfian parameters between 0 and 2. Each data set had 1 million elements, where the elements were integers drawn from the domain from 1 to 1 million. The code used for generating the data sets were also obtained from MassDal [95]. We also ran experiments on a Web crawl data set, originally obtained from Internet Archive [8], containing a stream of URLs sequentially extracting from the crawled pages. We hashed each URL in this collection to a 64-bit fingerprint, verified the data set and found no hash collisions between the URLs. The stream size (number of URL fingerprints) we used was 1 million. Using the second frequency moment of this URL stream, we approximately computed the Zipfian parameter assuming the URL frequencies follow the Zipfian model, and found that the Zipfian parameter were between 0.8 and 0.9. We also used longer and shorter stream sizes, but found similar Zipfian parameters and experimental results.

Experimental settings. In the experiments, we queried the multiplicities of all elements in the domain and the multiplicities of the top-100 frequent elements appeared in the data set using different sketching techniques. We obtained true frequencies of the elements using a sufficiently large buffer, and computed the absolute values of the differences between the estimates and the true frequencies as the error measurement.

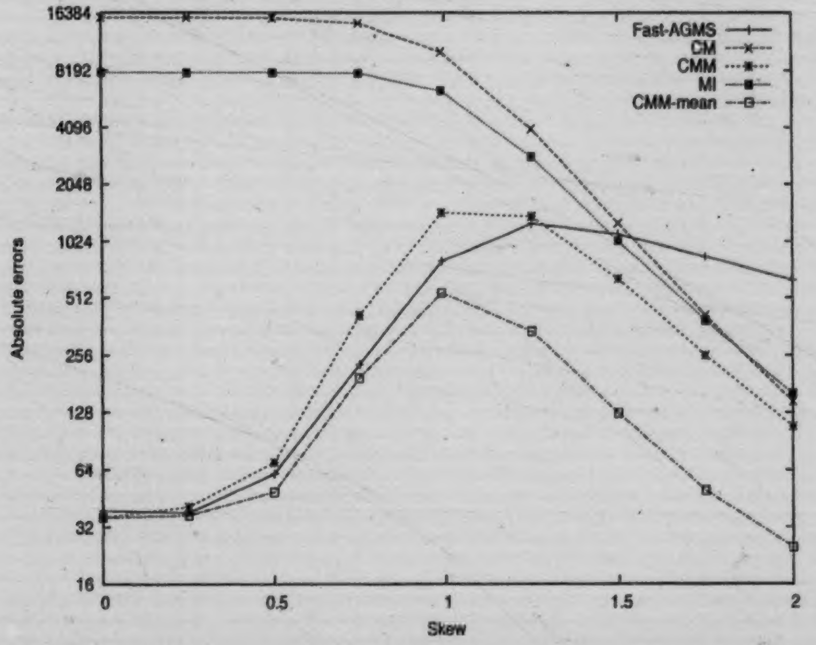


Figure 4.1: Average absolute errors vs. data set skew, comparing Fast-AGMS, CM, MI and our CMM; queries are all elements in the domain. The sketch width and depth are 64 and 3 respectively.

Varying the skew of the synthetic data sets. In our first experiment, we query each element in the domain once and return the average of the absolute errors of all queries. The result from data sets with different Zipfian parameters is shown in Figure 4.1. CMM-mean represents the algorithm using

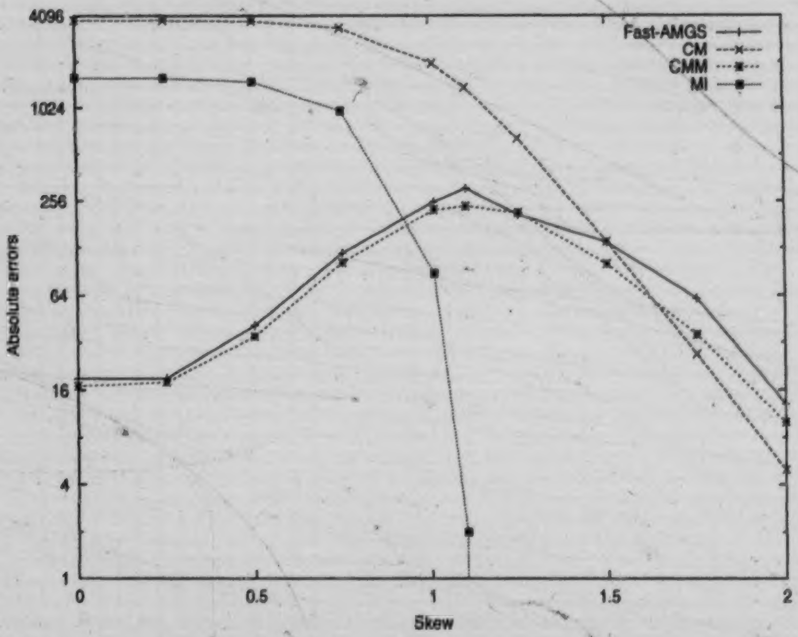


Figure 4.2: Average absolute errors vs. data set skew, for top-100 frequent element queries using Fast-AGMS, CM, CMM and MI. The sketch width and depth are 256 and 5 respectively.

the mean value of counters as the noise, while CMM represents the algorithm using the median of all counter values in a row as the noise. From the figure we can see that when the data set is less skewed, CMM-mean, Fast-AGMS and CMM all perform significantly better than CM and MI, while CM and MI become more accurate than Fast-AGMS when the data set is highly skewed.

Among CMM-mean, Fast-AGMS and CMM we also see some differences: CMM-mean and CMM both perform better than Fast-AGMS when the data set is highly skewed because of the CM bound applied to both CMM-mean and CMM; when the data set is less skewed, the performance of Fast-AGMS is between those of CMM-mean and CMM. Actually, CMM-mean performs well mainly because of the 0 bound we used. When the data set is skewed, it is very likely that there are some large outliers in row counters, which make CMM-mean significantly overestimate the noise, and accordingly return a negative estimate. This is good for those 0-frequency elements which do not appear

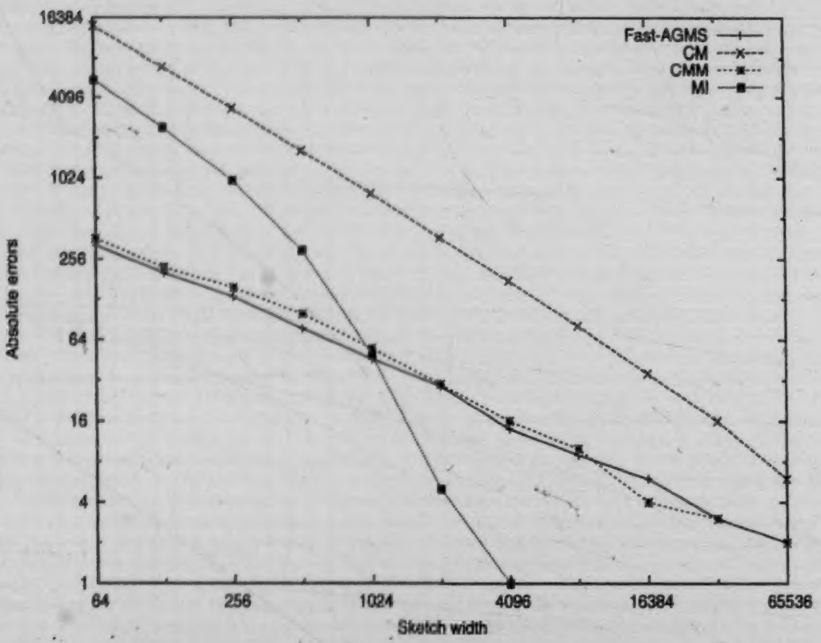


Figure 4.3: Average absolute errors vs. sketch width, for top-100 frequent element queries using Fast-AGMS, CM, CMM and MI on the 1M URL data set. The sketch depth is 5.

in the data set, because the final CMM-mean estimate will be 0 whenever CMM-mean returns a negative estimate. In contrast, CMM has less chance of overestimating the noise, thus CMM is less likely to take advantage of the 0 bound. Regarding Fast-AGMS, the chance of returning a negative estimate is one half for those 0-frequency elements. Given that a large fraction of query elements in the domain have frequency 0 in the synthetic data sets, which makes the 0 bound a dominant factor, in the rest of our experiments we focus on finding the multiplicities of frequent elements, where the 0 bound has much less impact on the experimental results.

In our second experiment, we query the multiplicities of the top-100 frequent elements. The average of the absolute errors of the 100 query answers on the data sets with different skew is shown in Figure 4.2. Some general trends observed from the figure are as follows. First, the accuracy difference between CMM and Fast-AGMS is small. Second, CMM and Fast-AGMS sig-

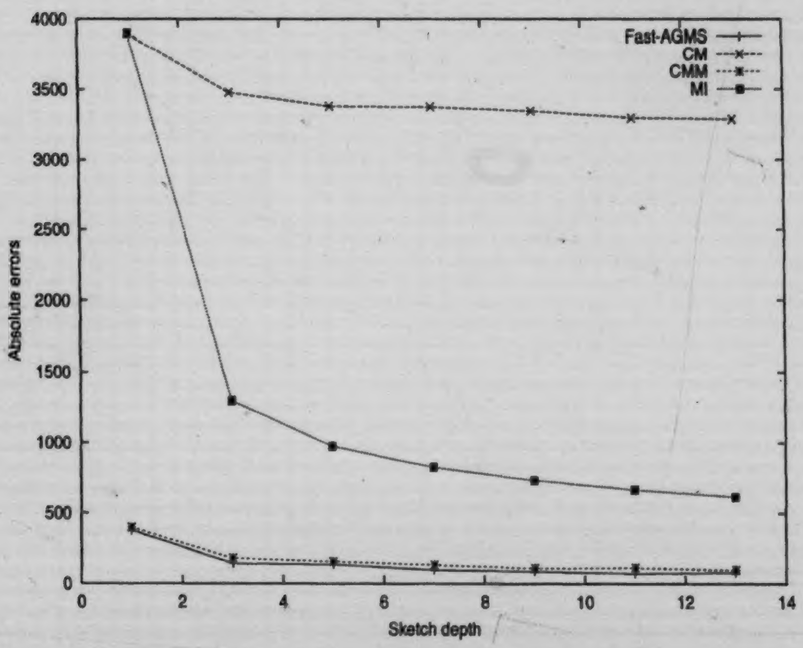


Figure 4.4: Average absolute errors vs. sketch depth, for top-100 frequent element queries using Fast-AGMS, CM, CMM and MI on the 1M URL data set. The sketch width is 256.

nificantly outperform CM when the data set is less skewed; the difference becomes smaller when the skew increases; when the data set is highly skewed, CM becomes more accurate than CMM and Fast-AGMS. Third, the MI heuristic consistently outperforms CM; but it is still much less accurate than CMM and Fast-AGMS for less skewed data; in the high skew cases, MI is much better.

One clear inconsistency between Figure 4.1 and 4.2 is the performance of MI. In the high skew cases, when query elements are the frequent ones, MI performs much better than all others, while MI performs much worse in Figure 4.1. This is because when the data set is highly skewed, there are less high frequent elements. The counters those elements are mapped to are very likely to be increased to a high value by the frequent elements themselves. When a non-frequent element arrives, it will only increase the minimum counters it is mapped to, which are less likely to be the ones frequent elements have touched because the values of those counters are likely to be very large already. Therefore, when the query elements are frequent ones, MI only gives very small errors. As discussed in Section 2.1.1, The benefit of this method depends on the frequency distribution and the order in which elements arrive. So it is difficult to be further analyzed.

Varying the sketch width on the real data set. In this set of experiments, we fixed the sketch depth to 5 and varied the sketch width. The results are shown in Figure 4.3. Similar to the results from the previous experiments, CMM performs very close to Fast-AGMS, and they both performs significantly better than CM. MI does not perform well when the space is small; but it becomes better when the space is large.

Varying the sketch depth on the real data set. In this set of experiments, we fixed the sketch width to 256 and varied the sketch depth. The results are shown in Figure 4.4. Similar to the results from the previous experiments, CMM performs very similarly to Fast-AGMS, and they both performs significantly better than CM. MI is better than CM, but not as good as CMM and Fast-AGMS.

4.2.5 Summary of Comparisons

In this chapter, we discuss 4 algorithms for approximately answering multiplicity queries: CM, Fast-AGMS, CMM and MI.

CMM, CM and Fast-AGMS. In general, CMM and Fast-AGMS give better estimates over a larger range of data sets. They perform similarly both in theory and in practice. But CM and CMM are 2 different estimation algorithms using exactly the same sketch. Therefore, the Count-min sketches can be more powerful than Fast-AGMS sketches as discussed in Section 4.1.

MI and other techniques. The MI heuristic consistently improves the accuracy of CM estimates, especially when the queries are frequent elements. In general, it may perform better than CMM and Fast-AGMS for highly skewed data sets when querying frequent elements. When the data set is less skewed, CMM and Fast-AGMS seem to perform better. But we are unable to reach a conclusion for our comparison because the results of MI may vary greatly even for data sets with the same skew but different element arrival orders. Furthermore, MI does not have certain nice properties, such as the ability to handle element deletions and the ease of analysis, which CMM, CM and Fast-AGMS all have. This is again because the arrival order of elements will change the performance of MI, while this order has no effect on CM, CMM and Fast-AGMS. In other words, the sketch will be the same for CM, CMM or Fast-AGMS as long as the frequencies of elements do not change, and hence the estimation will be the same regardless of the element order. Because MI is hard to be analyzed, the space bound remains the same as that of CM.

Time cost comparisons. The time cost of per element update for CM, CMM and Fast-AGMS is the same, i.e. $O(d)$ where d is the depth of the sketch. The time cost for MI depends on the number of hash functions used, and we are not sure how to set the number of hash functions properly to minimize the error.

As for the query time cost, CM needs $O(d)$ time to find the minimum counter. Both CMM and Fast-AGMS can find the median in $O(d)$ time as well using the SELECT algorithm [44], under the condition that the mean of all counters except the one touched by the query element is used to estimate the noise in CMM. But if CMM uses the median of each row for noise estimation, as we did in our experiments, then CMM needs $O(w)$ preprocessing time to find the median of counters for each sketch row. But those medians need to be computed only once and can be used for all queries.

4.3 Unbiased Self-join Size Estimates from Count-min Sketches

Count-min sketches can be also used to estimate the self-join size of a data stream as discussed in Section 2.1.1, where the estimate is an upper bound of the true value. Similar to the case of multiplicity queries, we propose a new estimation algorithm which gives an unbiased self-join size estimate of a data stream.

4.3.1 Our Estimation Algorithm

The CM algorithm [48] computes the sum of squares of all counters in each sketch row, and returns the minimum sum of all rows as the self-join size estimate. Our approach (CMM) use the same sketch with width w and depth d, but the estimation procedure is different: for each counter in a sketch row, we compute the average value of all other counters in the row except the counter itself, and deduct the average from that counter; by doing this, w residues are obtained, one for each counter. We then calculate the sum of the squares of the w residues, and return the product of the sum and $(w-1)/w$ as the self-join size estimate from that row. The final estimate is the median of the estimates from all d rows.

Formally, given a Count-min sketch $CM[0 \ldots d-1, 0 \ldots w-1]$ with d rows and w columns, we return the median of the following d values as the estimate:

$$\frac{w-1}{w}\sum_{j=0}^{w-1}(CM[i,j] - \frac{1}{w-1}(N - CM[i,j]))^2,$$

$$0 \leq i \leq d-1,$$

where N is the stream size, i is the row index and j is the counter index within a row. Next we show that this CMM algorithm gives an unbiased estimate for the self-join size and the variance is the same as that of AMS and Fast-AGMS.

4.3.2 Analyses of Our Algorithm

Lemma 4. *The estimate from each row of a Count-min sketch using the above CMM algorithm is expected to be the true self-join size (under pairwise independent hash functions), and the variance is* $\frac{4}{w-1}\sum_{x<y} f_x^2 f_y^2$ *(under 4-wise*

independent hash functions), where (x, y) is an arbitrary pair of distinct elements of the stream.

Proof. For a particular row of the sketch, let F_2 be the true self-join size, and $X_{x,j}$ be a Bernoulli random variable indicating if an element x is hashed to counter j $(j = 0, \ldots, w-1)$. That is

$$X_{x,j} = \begin{cases} 1, & x \text{ is hashed to the counter } j, (j = 0, \ldots, w-1); \\ 0, & \text{otherwise.} \end{cases}$$

Then the self-join size estimate

$$\begin{aligned}
\hat{F}_2 &= \frac{w-1}{w}\sum_{j=0}^{w-1}(\sum_x f_x X_{x,j} - \frac{1}{w-1}(N - \sum_x f_x X_{x,j}))^2 \\
&= \frac{1}{w(w-1)}\sum_{j=0}^{w-1}(w\sum_x f_x X_{x,j} - N)^2 \\
&= \frac{w}{w-1}\sum_{j=0}^{w-1}(\sum_x f_x^2 X_{x,j}^2 + 2\sum_{x<y} f_x f_y X_{x,j} X_{y,j}) \\
&\quad -\frac{1}{w-1}(2N\sum_x f_x \sum_{j=0}^{w-1} X_{x,j} - N^2) \\
&\quad (\because \sum_{j=0}^{w-1} X_{x,j}^2 = \sum_{j=0}^{w-1} X_{x,j} \equiv 1, \sum_x f_x = N) \\
&= \frac{w}{w-1}\sum_x f_x^2 + \frac{2w}{w-1}\sum_{x<y} f_x f_y \sum_{j=0}^{w-1} X_{x,j} X_{y,j} - \frac{N^2}{w-1} \\
&= \sum_x f_x^2 + \frac{1}{w-1}(\sum_x f_x^2 - N^2 + 2w\sum_{x<y} f_x f_y \sum_{j=0}^{w-1} X_{x,j} X_{y,j}).
\end{aligned}$$

The expected estimate

$$E[\hat{F}_2] \;=\; \sum_x f_x^2 + \frac{1}{w-1}(\sum_x f_x^2 - N^2$$

$$
\begin{aligned}
& +2w\sum_{x<y} f_x f_y \sum_{j=0}^{w-1} E[X_{x,j}X_{y,j}]) \\
& (\because \text{the hash function 2-wise independent,} \\
& \therefore E[X_{x,j}X_{y,j}] = \frac{1}{w^2}) \\
= \; & \sum_x f_x^2 + \frac{1}{w-1}((\sum_x f_x)^2 - N^2) = \sum_x f_x^2.
\end{aligned}
$$

The variance

$$
\begin{aligned}
& VAR[\hat{F_2}] = E[\hat{F_2}^2] - (E[\hat{F_2}])^2 \\
= \; & \frac{2}{w-1}(\sum_x f_x^2)E[\sum_x f_x^2 - N^2 \\
& +2w\sum_{x<y} f_x f_y \sum_{j=0}^{w-1} X_{x,j}X_{y,j}] + \frac{1}{(w-1)^2} \\
& E[(\sum_x f_x^2 - N^2 + 2w\sum_{x<y} f_x f_y \sum_{j=0}^{w-1} X_{x,j}X_{y,j})^2] \\
= \; & \frac{1}{(w-1)^2}E[(\sum_x f_x^2 - N^2)^2 \\
& +4w(\sum_x f_x^2 - N^2)(\sum_{x<y} f_x f_y \sum_{j=0}^{w-1} X_{x,j}X_{y,j}) \\
& +4w^2(\sum_{x<y} f_x f_y \sum_{j=0}^{w-1} X_{x,j}X_{y,j})^2] \\
= \; & \frac{1}{(w-1)^2}(2\sum_{x<y} f_x f_y)^2 - \frac{8}{(w-1)^2}(\sum_{x<y} f_x f_y)^2 \\
& +\frac{4w^2}{(w-1)^2}\sum_{x<y}(f_x^2 f_y^2 E[\sum_{j=0}^{w-1} X_{x,j}^2 X_{y,j}^2 \\
& +2\sum_{0\le j<k\le w-1} X_{x,j}X_{y,j}X_{x,k}X_{y,k}]) \\
& +\frac{8w^2}{(w-1)^2}\sum_{\substack{y_1<y_2 \\ x\neq y_1\neq y_2}}(f_x^2 f_{y_1} f_{y_2} E[\sum_{j=0}^{w-1} X_{x,j}^2 X_{y_1,j}X_{y_2,j} \\
& +\sum_{j\neq k} X_{x,j}X_{y_1,j}X_{x,k}X_{y_2,k}]) + \frac{24w^2}{(w-1)^2}
\end{aligned}
$$

$$
\begin{aligned}
& \sum_{x_1<x_2<y_1<y_2} (f_{x_1} f_{x_2} f_{y_1} f_{y_2} E[\sum_{j=0}^{w-1} X_{x_1,j} X_{y_1,j} X_{x_2} X_{y_2,j} \\
& + \sum_{j \neq k} X_{x_1,j} X_{y_1,j} X_{x_2,k} X_{y_2,k}]) \\
= & \frac{-4}{(w-1)^2} (\sum_{x<y} f_x f_y)^2 + \frac{4w}{(w-1)^2} \sum_{x<y} f_x^2 f_y^2 \\
& + \frac{8}{(w-1)^2} \sum_{\substack{y_1<y_2 \\ x \neq y_1 \neq y_2}} f_x^2 f_{y_1} f_{y_2} + \frac{24w^2}{(w-1)^2} \\
& \sum_{x_1<x_2<y_1<y_2} (f_{x_1} f_{x_2} f_{y_1} f_{y_2} (\frac{1}{w^3} + \frac{w(w-1)}{w^4}) \\
= & \frac{4}{w-1} \sum_{x<y} f_x^2 f_y^2 .
\end{aligned}
$$

□

Note that the variances of estimates from AMS [4, 3] and Fast-AGMS [46] are both $\frac{4}{w} \sum_{x<y} f_x^2 f_y^2$ given 4-wise independent hash functions. The difference between the expression of this variance and that of our CMM is in the terms w and $w-1$, meaning that CMM needs one more counter to reach the same variance. Since our CMM only needs one hash function per sketch row, while Fast-AGMS needs two per row and AMS needs w per row, CMM needs less space in storing hash functions. Thus, we consider the CMM variance the same as that of Fast-AGMS, and slightly smaller than that of AMS.

Theorem 6. *Let $\hat{F}_2$ be the self-join size estimate of a data stream using our CMM algorithm, and F_2 be the true self-join size. Given $O(log(1/\delta)/\epsilon^2)$ counters, with probability $1-\delta$, the relative error $|\hat{F}_2 - F_2|/F_2 \leq \epsilon$.*

Proof. This result is the same as that of AMS [4, 3] and Fast-AGMS [46] (the result for Fast-AGMS is shown in the form of join size of two data streams). Since in Lemma 4, we have shown that the variances of the estimates from these algorithms are all the same, the rest of the proof is just applying Chebyshev's Inequality. Details can be found in [4]. □

4.3.3 Experiments for Self-join Size Estimations

To verify the performance of CMM in estimating self-join sizes, we ran two sets of experiments comparing CMM with CM and Fast-AGMS. Since AMS needs to update all sketch counters for each element, which is too slow for many real-time data stream applications, and Fast-AGMS is a much faster but similar alternative with the same estimation expectation and variance, we do not include AMS in our experiments.

Experimental settings. For each sketch row, we computed a self-join size estimate using CMM, Fast-AGMS and CM respectively. Then for CMM and Fast-AGMS, we return the median of estimates obtained from all sketch rows; for CM, we return the minimum value of estimates obtained from all sketch rows.

The data sets used in this experiments and the sketch construction process were the same as in the multiplicity query experiments described in Section 4.2.4. Cormode and Muthukrishnan [49] propose a variation of the CM algorithm, called CM-, for less skewed (Zipfian parameter < 1) and uniform data sets. Their experiments on data sets similar to ours shows that CM and CM- performs similarly, hence we did not include CM- in our experiments.

Varying data set skew. In this experiment we fixed the sketch width and depth and varied the skew of the synthetic data sets. The results are shown in Figure 4.5. The two sub-figures are the same except that the second one shows a small error range so that the difference between CMM and Fast-AGMS can be seen.

From the figure we can see that when the data set is low skewed, CMM and Fast-AGMS perform significantly better than CM; when the data set is more skewed, the difference becomes smaller. Furthermore, the difference between CMM and Fast-AGMS is always small.

Varying the sketch width. In this experiment, we fixed the sketch depth varying the sketch width and ran our experiments on the URL fingerprint data set. The results are shown in Figure 4.6. From the sub-figures we can see that CMM and Fast-AGMS always perform similarly, and they both outperform

CM significantly especially when the space is small.

Varying the sketch depth. In this experiment, we fixed the sketch width varying the sketch depth and ran our experiments on the URL fingerprint data set. The results are shown in Figure 4.7. Again, from the figure we can see that CMM and Fast-AGMS perform similarly, and they both outperform CM significantly. Furthermore, increasing the sketch depth within a small range (e.g. from 1 to 10) has almost no impact on the estimation accuracy for all tested algorithms. Because of the time cost, exponentially increasing the sketch depth is infeasible in most real-time applications.

Time cost comparisons. The time efficiencies for CMM, CM and Fast-AGMS are the same. In terms of per element update, CMM, CM and Fast-AGMS all need $O(d)$ time. Regarding the query answering time, the costs of these methods are still the same: they all scan counters in a sketch row linearly, i.e. $O(w)$ time; in CMM and Fast-AGMS, finding the median of estimates from all rows requires $O(d)$ time using the SELECT algorithm [44]; finding the minimum value of the counters in CM also requires the same time.

4.4 Related Work

There are many data stream summary techniques, each proposed for different purposes. In this section, we only discuss the work closely related to ours and not covered earlier in this thesis.

Krishnamurthy et al. independently proposed a technique called k-ary sketch [86], which is similar to our CMM technique. But their goal is to detect changes for IP packet streams, and they do not compare their technique with Fast-AGMS or Count-min. Our explanation is also much simpler than theirs. Most importantly, we proposed the median heuristic such that the estimation errors are significantly decreased to a similar level to those from Fast-AGMS. Accordingly, we can claim that our CMM method is more powerful and flexible than Fast-AGMS.

Finding frequent elements. There are some work (e.g. [94, 84, 97]) focusing on finding frequent elements approximately in a data stream. These algorithms also construct data summaries in one pass, but they are specialized for finding frequent items and not for other queries.

Recent applications of Count-min and Fast-AGMS. Korn et al. [85] use Count-min sketches as underlying data structures to answer multiplicity queries, self-join size estimations, range sum queries, quantile approximations. Cormode and Garofalakis [46] apply Fast-AGMS in a distributed environment to answer multiple queries such as multiplicity queries, iceberg queries, range queries, join and self-join size estimations. Indyk and Woodruff [81] use Fast-AGMS as a building block to find the k-th ($k > 2$) frequency moments.

4.5 Summary and Potential Extension

In this chapter, we propose new estimation algorithms, CMM, for multiplicity queries and self-join size estimations based on a data stream summary technique, Count-min. Compared to the previous estimation algorithms based on Count-min, our new methods significantly improve the estimation accuracy on a wide range of data sets. In contrast with another influential general-purpose data stream summary technique Fast-AGMS, Count-min sketches can give estimates with the same accuracy, time and space efficiency using CMM. Moreover, there are other attractive estimation options and error bounds for Count-min, which are not applicable to Fast-AGMS; with our new estimation algorithms, we make a case that Count-min is more flexible and powerful.

In addition to the applications of finding the top-k frequent elements and answering iceberg queries, CMM can be potentially extended to answer other queries such as SUM aggregates (i.e. a generalization of multiplicity queries where frequency updates are not limited to 1 and -1), range queries, quantiles approximations and join size estimations, as shown by Cormode and Muthukrishnan [48] for CM. Some of these extensions are straightforward, but others need further research.

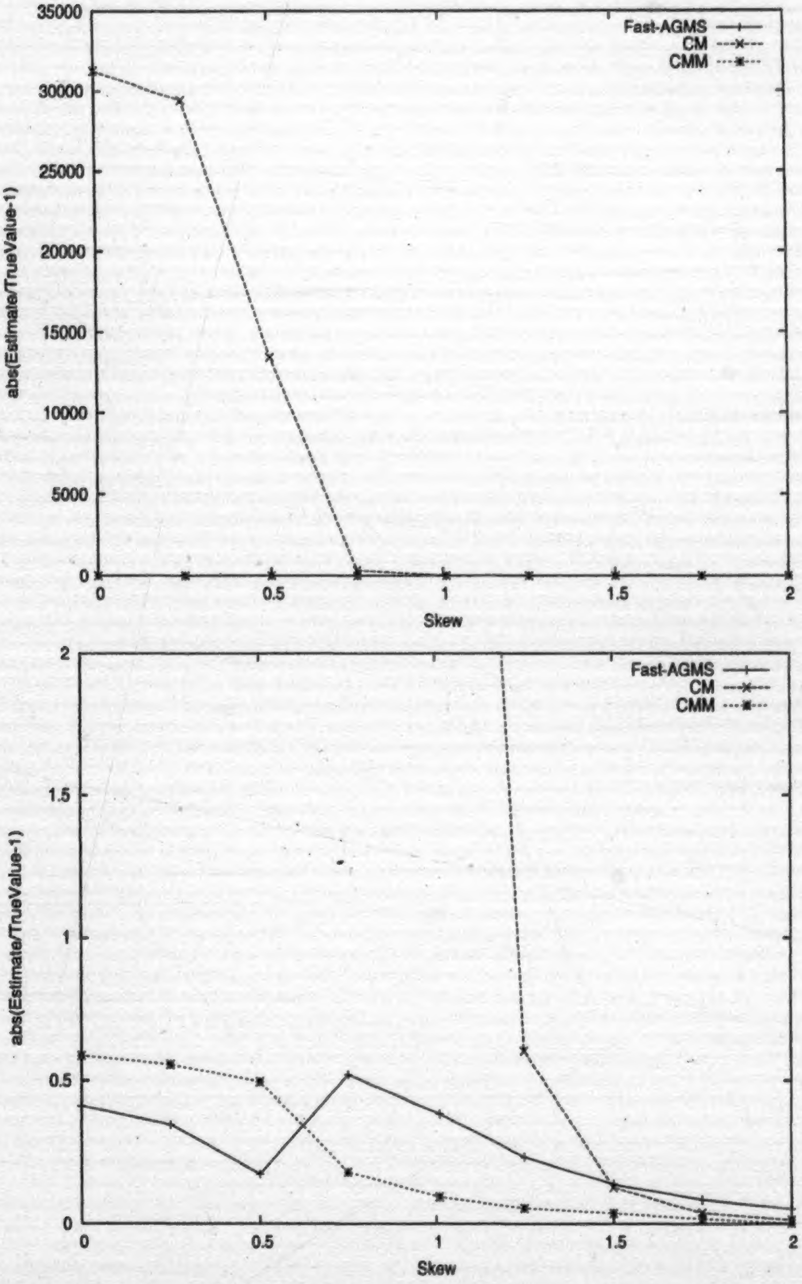


Figure 4.5: Self-join size estimation errors vs. data set skew, comparing Fast-AGMS, CM and CMM. The sketch width and depth are 16 and 5 respectively. (The 2nd sub-figure zooms in the 1st one). The stream sizes are all 1 million.

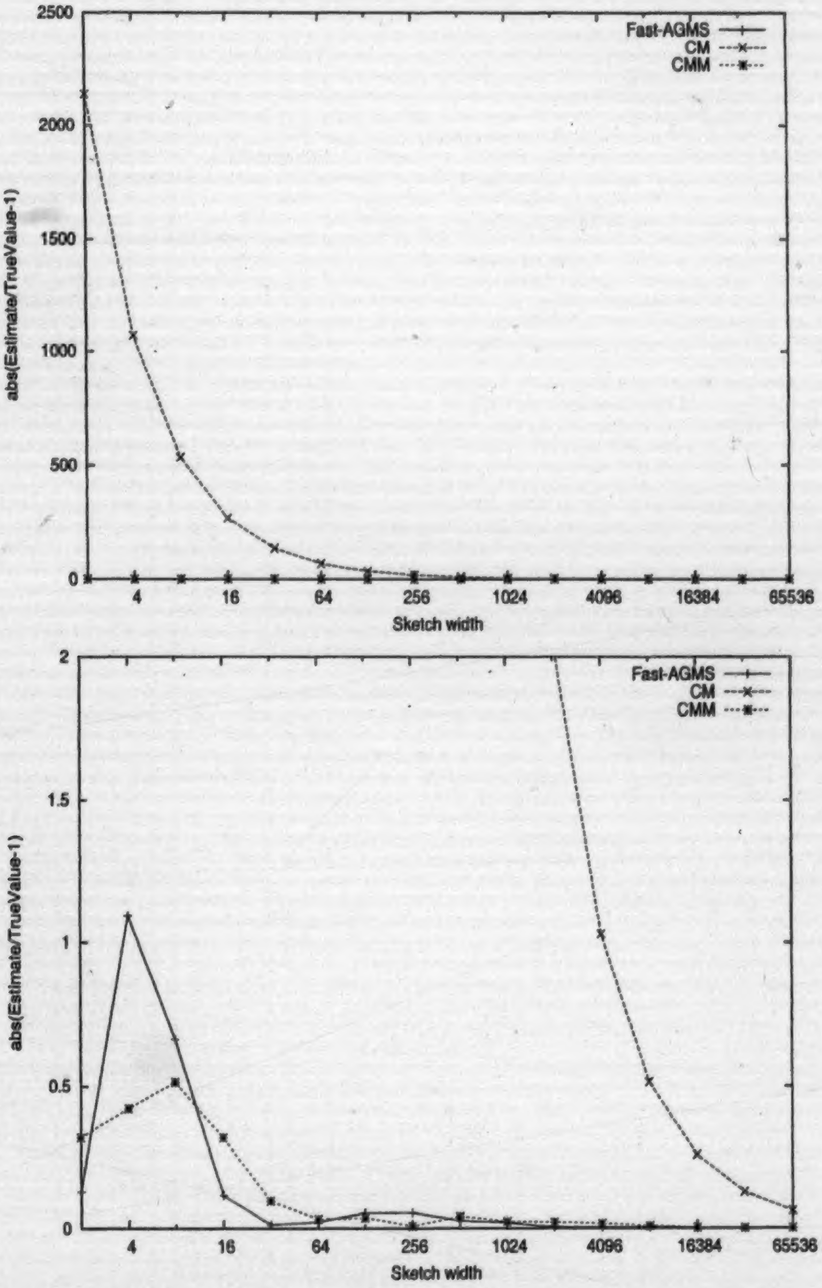


Figure 4.6: Self-join size estimation error vs. sketch width, comparing Fast-AGMS, CM and CMM on the 1M URL data set. The sketch depth is 3. The 2nd sub-figure zooms in the 1st one.

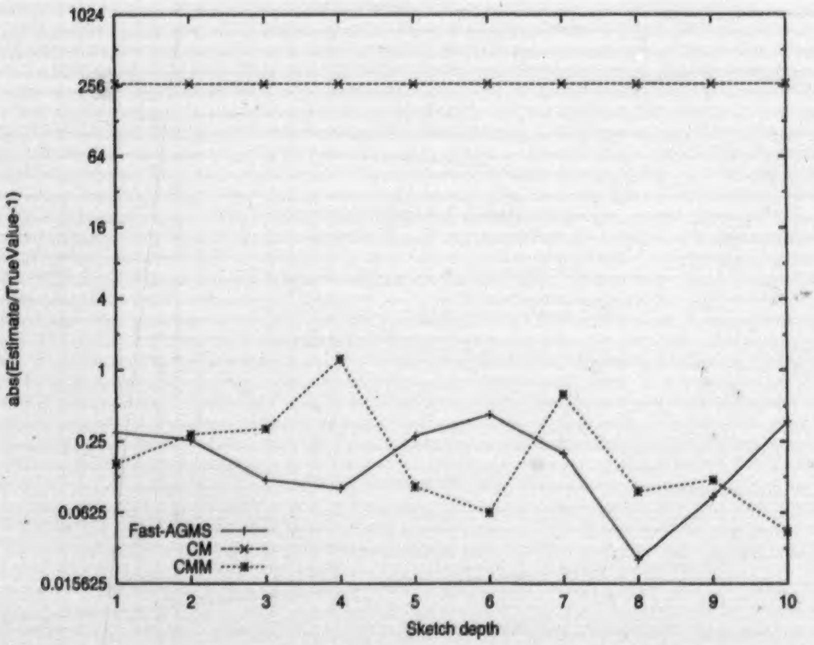


Figure 4.7: Self-join size estimation error vs. sketch depth, comparing Fast-AGMS, CM and CMM on the 1M URL data set. The sketch width is 16.

Chapter 5

Efficient Result Set Size Estimation for Similarity Queries on Large Data Sets

5.1 Similarity Join/Self-join and Selectivity Estimation

Similarity queries are applied to a wide range of domains with applications ranging from finding near-duplicate Web pages, filtering approximately duplicate records, detecting possible plagiarism, detection of similar protein structures, etc. In many of these applications, data must be self-joined before near-duplicates can be listed. Consider for instance cleaning and filtering in a data integration environment where data records are gathered from multiple different sources and the same entity can be described differently, leading to both redundancies and inconsistencies. For example, multiple records can refer to the same person, address, product description, etc. Inconsistencies may also arise due to misspellings and encoding differences. *Similarity join* is considered as a key operation in reconciling many of these inconsistencies and has received more attention recently. The work includes efficient near-duplicate detection [77, 19, 59], set similarity join [7, 34] and finding fuzzy duplicates [36, 32].

Efficiently evaluating similarity join and self-join on large datasets can be computationally challenging. For example, in a Web document clustering application, Broder et al. [22] spent more than 10 CPU days to find all 50% similar pairs among 30 million documents; the memory requirement of the work was 20GB. Although hardware is getting faster and memory is becoming larger, the Web is also growing in the same or even larger scale. A similar experiment was run more recently by Henzinger on 1.6 billion pages but the running time of the experiment was not reported [77]. A reason for this behaviour is that their near-duplicate detection algorithms have a few key steps which are not parallelizable [22], hence their powerful distributed cluster is not fully utilized. Despite these and other algorithmic research work in the area, the nature of the problem determines that it can be very expensive both in time and space cost when the result set is large, no matter how good the algorithms are. In many cases, knowing an estimate of the result set size before actually executing the potentially expensive operation is important.

In this chapter, we study the problem of estimating the result set sizes of two types of queries: similarity self-join/join and similarity search. Here are a few motivating examples for this work.

- Estimating result set is important in building more interactive and user-friendly systems. With an estimate of the result set and the portion of work done so far, one may predict the approximate remaining work and the expected wait time, and pass this information to users (e.g. in the form of a progress bar).

- With the current and expected level of support for similarity queries in DBMSs, estimating the selectivity of similarity predicates is important for estimating the costs of query plans and generally for optimizing queries.

- In data cleaning and filtering over large datasets, knowing the result set sizes before the actual operations is desired and can be helpful. For example, if there are hundreds or thousands of tables, the data cleaner should know which one is 'dirty' and which one is 'clean'. This is particularly important for very large tables; if the estimates show that a table is clean, meaning that it doesn't have many near-duplicates, an expensive data cleaning task may be avoided.

Clearly, the techniques for estimating a query result set size should be inexpensive and accurate. The estimation process should be much faster than running the queries; otherwise it is not much useful. Often an approximate fast answer is preferred over a slow exact answer. However, the approximate estimates should be relatively accurate; fast but wrong estimates are useless at all. The inexpensiveness and accuracy requirements make the result set size estimation a challenging problem for large data sets.

In this chapter, we focus on estimating the result sizes of the aforementioned queries (similarity self-join, join and similarity search) when the similarity measure is based on the Hamming distance. Specifically under this distance, two vectors are similar if they have certain number of coordinates in common, and a pair of records are similar if some fraction of their columns are the same. This similarity measure has been useful in several different applications.

For example, Broder et al. [19, 22] and Henzinger [77] fingerprint web documents (or strings) into 6-dimensional vectors, which they call super-shingls, by using min-wise independent hashing [21, 20] on *n*-grams generated from documents. One nice property of this technique is that the number of common coordinates between the hashed vectors of two documents is expected to give the Jaccard similarity [1] of the two documents. As another example,

---

[1] Jaccard similarity measure of two set of terms is the size of intersection of the two sets divided by the size of the union of the two sets

Charikar [29] shows that documents (or strings) can be sketched using a random projection into vectors such that the number of common coordinates is proportional to the Cosine similarities [2] of the documents. Henzinger [77] also uses this technique in her experiments for finding near-duplicate documents. In data cleaning applications, the Hamming distance between, for example, the records of a *Person* table with 6 attributes *(First Name, Last Name, DateOfBirth, Address, HomePhoneNumber, CellPhoneNumber)* would indicate how many record pairs share 1, 2, ..., 6 attributes. If two records have certain number of common attributes, it is very likely that they are duplicates.

Our contributions in this Chapter. In this chapter, we propose efficient probabilistic algorithms for approximating the result set size of similarity queries: similarity self-join, join and search. To the best of our knowledge, there is no prior work addressing the same issue, giving unbiased estimates and tight error bounds, in one pass over data. We consider both offline and online scenarios for processing the queries. By offline we mean the amount of space available for storing the data summary is large; by online we mean only a small amount of memory is available for the data summary. The basic idea of our algorithms is as follows: if two vectors share certain number of coordinates, when we randomly pick a few coordinates, which we call their concatenation a super-value, from both vectors, the chance of selecting the same super-value is proportional to the similarity of the two vectors. By repeating this random selection process for each vector in the data set and finding statistics of the super-value streams, we can estimate the number of similar vector pairs. In other words, we map the result set size estimation problems for multidimensional similarity queries to frequency estimation problems for one dimensional data streams, where sketching techniques can be used to efficiently estimate join/self-join sizes and frequency counts of the data streams using a very small amount of space.

---

[2] Cosine similarity of two documents is measured by the angle between the two attribute vectors of the documents. In information retrieval, document attribute vectors are usually TF-IDF vectors of the document, where TF indicates the frequency of a term in a document and IDF indicates the popularity of the term in the document collection.

Both theoretical analysis and experimental evaluation show that our algorithm works well for large data sets. Specifically, the absolute errors for the result set size estimates are only related to the true answer, independent of the data set size (the number of vectors in the data set); the relative error is inversely proportional to the true answer. As for the selectivity estimation, errors generated from our algorithms are inversely proportional to the data set size. Thus, our algorithms are efficient and accurate for large data sets.

We believe a linear scan of the data set is not a strict requirement even for very large data sets. Our data summary structure can be incrementally constructed whenever a vector or record is inserted into the data set. Also, our algorithms are fully parallelizable. A large data set can be cut into small pieces and distributed over multiple machines to speed up the process; a centralized data summary can also be easily constructed from summary fragments on different machines, thus estimates for the overall data set can be generated. However, we have to point out that our algorithms may not be applicable to high-dimensional vectors, although they work well in the kind of examples we have mentioned here.

Outline of this chapter. Section 5.2 discuss our PairCount algorithm for similarity self-join size estimations, which is the core of all other algorithms; Section 5.3 extends PairCount to estimate the result set size of similarity join and search; experimental results are reported in Section 5.5. Section 5.4 discusses how our algorithms can be extended to handle other similarity measure, and a few potential applications are list. Section 5.6 is related work. Last, we summary this chapter and discuss possible extension of this work.

5.2 Similarity Self-join Size Estimation

This section presents the problem statements and our proposed algorithms in both offline and online scenarios.

5.2.1 Problem Statement

We first give the definitions of some of the terms we use in this section.

Definition 3. *(Similarity measure) A pair of d-dimensional vectors is* ***s-similar*** *if and only if the two vectors have s coordinates in common. Clearly, this similarity measure is based on Hamming distance.*

Definition 4. *(Similarity self-join size and selectivity) For a set of n vectors, the s-similarity self-join size is the number of vector pairs that are at least s-similar; the selectivity of s-similarity self-join is the fraction of vector pairs that are at least s-similar among all possible pairs.*

Problem 1. *(Similarity self-join size and selectivity estimation) Given n d-dimensional vectors where each coordinate of the vectors is a non-negative number, our goal is to efficiently estimate the size and selectivity of an s-similarity self-join, where s is a given parameter. The problem can be also defined similarly for a relation with n records and d attributes; since each attribute value can be fingerprinted into a non-negative number if necessary, we do not distinguish vectors or records in this chapter.*

Based on different potential applications, we consider two scenarios: offline and online. By offline we mean the application does not require an immediate query answer right after the data processing; intermediate data summary can be flushed to disks. By online we mean the application requires timely query result on-the-fly once the input data has been processed. Usually data can be processed in memory in a streaming fashion. We first discuss our proposed algorithms for the offline case. The online algorithm is similar to the offline one in general except that the space consumption must be bounded such that the intermediate results can fit in memory.

5.2.2 A Straightforward Solution

For a relatively small data set (e.g. less than 100,000 records), the problem can be solved precisely in the following way: store all n records into a list in memory; compare each record i with all other records that appear after i in a given ordering, and obtain their similarity by counting the number of common attributes; for each pair of s-similar records, increase the counter $C[s]$ by one. The number of comparison is $O(n^2)$, and the space cost is $O(n)$. Clearly, this algorithm is inefficient for large data sets. For example, it takes about 5 minutes to process 100,000 records with 6 attributes each on a AMD64/4000 machine, while processing 400,000 similar records takes more than one hour on the same machine. For large data sets containing millions of records, this algorithm is generally too slow. If the data set does not fit in memory, the situation becomes even worse, and the algorithm may not be feasible at all.

5.2.3 Random sampling

A widely used approximation technique is random sampling. For the similarity self-join size estimation problem, random sampling is also applicable. One can pick R different records from the input data set uniformly at random (sampling without replacement); then use the straightforward algorithm to find the similarity self-join size of the sample set; last, scale the similarity self-join size of the sample set by a factor of $\frac{n(n-1)}{R(R-1)}$. Alon et al. [3] used a similar random sampling technique in their experiments for estimating the self-join sizes of data streams. However, the results show that it is not as accurate as other methods. Although random sampling has certain nice properties such as the ability of handling different queries and the implementation easiness, it may not be the best choice for specific tasks as a general-purpose data summary. We compared random sampling with our method in our experiments and showed that our method is significantly more accurate than random sampling in the experiments section of this chapter.

5.2.4 Our SelfJoinPairCount Algorithm (Offline Scenario)

Given a data set with n records, each having d attributes, our offline *SelfJoinPairCount* algorithm first scans the data set once and constructs data summaries (a few data streams). Based on the self-join size of those streams, SelfJoinPairCount estimates the similarity self-join size.

Step 1 - Transforming input data into super-value streams. SelfJoinPairCount scans the data set once. For each record and each $k = s, \dots, d$, SelfJoinPairCount picks k different attribute values uniformly at random, tags each attribute value with its attribute index, concatenates the k labeled values into one *k-super-value* in their attribute index order [3]; repeat this process $l_k = \lceil r\binom{d}{k} \rceil$ times for $0 < r \leq 1$, and store all k-super-values as a stream of items either in memory or on disks.

Step 2 - Finding the self-join size of super-value streams. In the offline scenario, obtaining the precise self-join size for a data stream can be done efficiently. If the stream fits in memory, a hash index can be used to speed up the process of detecting duplicates and counting the frequency of each distinct item. Accordingly, the self-join size can be computed easily. If the stream does not fit in memory, external sorting needs $O(log^{R}_{M-1})$ passes to sort the data stream, where M is the number of memory pages available, and R is the number of blocks occupied by the data stream. Having a sorted stream, computing the self-join size is straightforward.

Step 3 - Estimating the similarity self-join size. Let Y_k denote the self-join size of the k-super-value stream found in the previous step; let X_k be the estimated number of k-similar record pairs, and x_k be the true value. To find out the s-similarity self-join size, $g_s = \sum_{k=s}^{d} x_k$, SelfJoinPairCount computes each X_k and accordingly $G_s = \sum_{k=s}^{d} X_k$ using the following formula:

$$X_k = (Y_k - l_k n - \sum_{j=k+1}^{d} \mu_{j,k} X_j)/\mu_{k,k}, \tag{5.1}$$

where $\mu_{j,k} = l_k^2 \binom{j}{k} / \binom{d}{k}^2$ is the expected value that a j-similar pair contributes to Y_k, $k \in [s, d], j \in [k, d]$.

---

[3] If necessary, a k-super-value can be fingerprinted into a fixed length hash value. Provided a proper fingerprint size, the mapping can be guaranteed to have almost 0 hash collision.

Algorithm analysis. The basic idea of the algorithm is as follows: two records have certain chance of generating the same k-super-value if they are at least k-similar; thus such record pairs are expected to contribute a certain number of duplicates to the k-super-value stream. By checking the self-join size of those streams and estimating the contribution from record pairs with different similarities, we can estimate the number of pairs that are $(d, \ldots, k)$-similar.

Next, we discuss the property and performance of this offline SelfJoinPairCount algorithm including its accuracy, time and space costs.

Theorem 7. *(Unbiased similarity self-join size estimate and the standard deviation - offline case) The offline SelfJoinPairCount algorithm gives an unbiased estimate of the s-similarity self-join size, i.e.* $E[G_s] = g_s$*, and the standard deviation of* $\frac{G_s}{g_s}$ *is at most*

$$\binom{d}{s}\sqrt{\frac{1}{r}\binom{2(d-s)}{d-s}}/g_s,$$

where G_s *is the estimate and* g_s *is the true value.*

Proof. Let $Z_{o_j,k}$ be the value that a j-similar record pair, denoted by o_j, contributes to Y_k, the self-join size of the k-super-value stream.

The main structure of this proof is to find the expected values and variance bounds of $Z_{o_j,k}$, Y_k, X_k and G_s sequentially. In the middle of this process, we use a simplified expression of X_k, which is a function of Y_j ($k \leq j \leq d$) rather than a function X_j ($k+1 \leq j \leq d$).

First, we have [4]

$$Y_k = \sum_{j=k}^{d} \sum_{\forall o_j} Z_{o_j,k} + nr\binom{d}{k}. \tag{5.2}$$

Note that X_k, Y_k and $Z_{o_j,k}$ are all random variables. The expected value of

---

[4] Since $r\binom{d}{k}$ may not be an integer, in practice SelfJoinPairCount picks $l_k = \lceil r\binom{d}{k} \rceil$ k-combinations, but in our analysis we assume $l_k = r\binom{d}{k}$, and this should not significantly affect our results.

$Z_{o_j,k}$

$$\mu_{j,k} = E[Z_{o_j,k}] = l_k \frac{\binom{j}{k}}{\binom{d}{k}} \frac{l_k}{\binom{d}{k}} = r^2 \binom{j}{k}.$$

$$\therefore E[Y_k] = r^2 \sum_{j=k}^{d} \binom{j}{k} x_j + nr \binom{d}{k}.$$

$\mu_{j,k}$ is the expected value that a j-similar pair contributes to Y_k. From Eq. 5.1 we can see that SelfJoinPairCount removes the contributions of $\{k+1, k+2, \ldots, d\}$-similar pairs from Y_k, thus it is not hard to verify that X_k is an unbiased estimate for x_k. Next, we will focus on the proof of the standard deviation part.

Let $p_{j,k,i}$ be the probability that a j-similar record pair contributes i to Y_k, then the variance of $Z_{o_j,k}$

$$\begin{aligned}
\sigma^2_{j,k} =& VAR[Z_{o_j,k}] = E[Z^2_{o_j,k}] - \mu^2_{j,k} = \sum_{i=1}^{l_k} i^2 p_{j,k,i} - \mu^2_{j,k} \\
\leq & l_k \sum_{i=1}^{l_k} i p_{j,k,i} - \mu^2_{j,k} = l_k \mu_{j,k} - \mu^2_{j,k} \\
=& r^3 \binom{j}{k} \binom{d}{k} - r^4 \binom{j}{k}^2 \leq r^3 \binom{j}{k} \binom{d}{k}.
\end{aligned}$$

Second, we prove by induction that (the simplified expression of X_k)

$$X_k = \frac{1}{r^2} \sum_{j=k}^{d} (-1)^{j-k} \binom{j}{k} Y_j + C_k, \tag{5.3}$$

where $k \in [1, d]$, and C_k is a constant which is not important in our analysis. From Eq. 5.1, we can easily verify that Eq. 5.3 holds for $k = d, d-1$. Assuming Eq. 5.3 holds for k, we need to prove it holds for $k-1$ as well.

From Eq. 5.1 we have

$$
\begin{aligned}
X_{k-1} &= (Y_{k-1} - l_{k-1}n - \sum_{j=k}^{d} \mu_{j,k-1} X_j)/\mu_{k-1,k-1} \\
&= \frac{1}{r^2} Y_{k-1} - \frac{l_{k-1}n}{r^2} - \sum_{j=k}^{d} \binom{j}{k-1} X_j \\
&= \frac{1}{r^2} Y_{k-1} - \frac{l_{k-1}n}{r^2} - \sum_{j=k}^{d} \binom{j}{k-1} (\frac{1}{r^2} \sum_{i=j}^{d} (-1)^{i-j} \binom{i}{j} Y_i + C_j) \\
&= \frac{1}{r^2} Y_{k-1} - \frac{1}{r^2} \sum_{i=k}^{d} \sum_{j=k}^{i} (-1)^{i-j} \binom{j}{k-1} \binom{i}{j} Y_i + C_{k-1} \\
&= \frac{1}{r^2} Y_{k-1} - \frac{1}{r^2} \sum_{i=k}^{d} \binom{i}{k-1} \sum_{j=k}^{i} (-1)^{i-j} \binom{i-k+1}{j-k+1} Y_j + C_{k-1} \\
&= \frac{1}{r^2} Y_{k-1} - \frac{1}{r^2} \sum_{i=k}^{d} \binom{i}{k-1} Y_i (-1)^{i-k} + C_{k-1} \\
&= \frac{1}{r^2} \sum_{i=k-1}^{d} (-1)^{i-k+1} \binom{i}{k-1} Y_i + C_{k-1}.
\end{aligned}
$$

$\therefore$ Eq. 5.3 holds for $k-1$, thus it holds for all $k \in [1, d]$.

$$
\begin{aligned}
\therefore G_s = \sum_{k=s}^{d} X_k &= \frac{1}{r^2} \sum_{k=s}^{d} \sum_{j=k}^{d} (-1)^{j-k} \binom{j}{k} Y_j + \sum_{k=s}^{d} C_k \\
&= \frac{1}{r^2} \sum_{j=s}^{d} \sum_{k=s}^{j} (-1)^{j-k} \binom{j}{k} Y_j + \sum_{k=s}^{d} C_k.
\end{aligned}
$$

Last, assuming the covariance between Z_{o_i,k_1} and Z_{o_j,k_2} is small, we have

$$
VAR[Y_k] \approx \sum_{j=k}^{d} x_j \sigma_{j,k}^2 \le r^3 \binom{d}{k} \sum_{j=k}^{d} \binom{j}{k} x_j.
$$

$$
\begin{aligned}
\therefore VAR[G_s] &\le \frac{1}{r^4}\sum_{k=s}^{d}\binom{k}{s}^2 VAR[Y_k]\\
&\le \frac{1}{r}\sum_{k=s}^{d}\binom{k}{s}^2\sum_{j=k}^{d}x_j\binom{j}{k}\binom{d}{k}\\
&= \frac{1}{r}\sum_{k=s}^{d}\sum_{j=k}^{d}x_j\binom{j}{s}\binom{j-s}{k-s}\binom{d}{s}\binom{d-s}{k-s}\\
&\le \frac{1}{r}\binom{d}{s}^2\sum_{k=s}^{d}\binom{d-s}{k-s}^2\sum_{j=k}^{d}x_j \le \frac{1}{r}\binom{d}{s}^2\sum_{j=s}^{d}x_j\sum_{k=s}^{d}\binom{d-s}{k-s}^2\\
&= \frac{1}{r}\binom{d}{s}^2\binom{2(d-s)}{d-s}g_s.
\end{aligned}
$$

Therefore, the standard deviation of $\frac{G_s}{g_s}$ is at most

$$
\binom{d}{s}\sqrt{\frac{1}{r}\binom{2(d-s)}{d-s}}/g_s.
$$

□

Remarks. This result shows that the estimation accuracy is dominated by several factors: the true similarity self-join size g_s, the number of attributes d, the given similarity threshold s, and the sampling rate r. In general, a larger data set has a larger g_s. For fixed d, s and r, the relative error is expected to decrease when n increases. Therefore, SelfJoinPairCounts is suitable for large data sets.

In the expression of the standard deviation, s and d are also important factors. When d and s are small, the standard deviation is small; but when d increases, the expected error increases sharply. For example, when $d = 5$ and $s = 4$, $\binom{d}{s}\sqrt{\binom{2(d-s)}{d-s}} = 7$; when $d = 10$ and $s = 8$, $\binom{d}{s}\sqrt{\binom{2(d-s)}{d-s}} = 110$; when $d = 20$ and $s = 16$, $\binom{d}{s}\sqrt{\binom{2(d-s)}{d-s}} = 40536$.

Meanwhile, d also affects the selection of r. For each record, SelfJoinPairCount generates $r\sum_{k=s}^{d}\binom{d}{k}$ super-values, which dominates the time and space costs. To keep the time or space cost reasonably small, r should not be a large value. In general, depending on the data set size and the given similarity threshold, SelfJoinPairCount work well only when the number of attributes is relatively small.

Next theorem shows the space and time costs to bound the selectivity error. Note that we just discuss the time and space cost for processing the data set; the cost for generating the result, e.g. finding the self-join size of super-value streams, depends on if the data set fits in memory or not, and it is not hard to see based on the description in our algorithm.

Theorem 8. *(Space and time cost to bound the selectivity error) The offline SelfJoinPairCount algorithm guarantees that the estimated selectivity of the similarity self-join deviates from the true value by at most* ϵ *with probability at least* $1-\lambda$. *More precisely,* $Pr[|\hat{\theta}_s - \theta_s| \leq \epsilon] \geq 1-\lambda$, *where* $\hat{\theta}_s$ *is the estimated selectivity and* θ_s *is the true value. The space cost for processing the input data set is* $O(log(1/\lambda)\binom{d}{s}^2\binom{2(d-s)}{d-s}2^d/(\epsilon^2 n))$, *and the time for processing each record is* $O(log(1/\lambda)\binom{d}{s}^2\binom{2(d-s)}{d-s}2^d/\epsilon^2 n^2)$ [5], *where* n *is the number of records,* d *is the number of attributes, and* s *is the given similarity threshold.*

Proof. From Theorem 7 we know the variance of the similarity self-join size, $VAR[G_s] \leq \frac{1}{r}\binom{d}{s}^2\binom{2(d-s)}{d-s}g_s$. By Chebyshev Inequality we have

$$Pr[|G_s - g_s| > \epsilon n^2] \leq \frac{VAR[G_s]}{\epsilon^2 n^4} \leq \frac{\binom{d}{s}^2\binom{2(d-s)}{d-s}}{r\epsilon^2 n^2}.$$

Let the last term above be a constant, say $\frac{1}{8}$, we have

$$r = 8\binom{d}{s}^2\binom{2(d-s)}{d-s}/(\epsilon^2 n^2).$$

To increase the success probability, we can repeat the same algorithm independently $2log(1/\lambda)$ times, and take the median of the multiple results. Due to

---

[5] *when this term is smaller than* $O(1)$, *the time bound is not valid because each* $l_k \geq 1$. *We have this result because we assume* $l_k = r\binom{d}{k}$ *in our analysis, while in practice* $l_k = \lceil r\binom{d}{k}\rceil$. *When* $l_k = 1$, *the time and space cost is clear, and we do not discuss this case in the following theroems.*

Chernoff bound [99], we can guarantee the probability that SelfJoinPairCount fails is at most λ. Since SelfJoinPairCount picks $O(r2^d)$ super-values for each record, the time and space costs stated in the theorem can give the desired result. □

Remarks. This theorem shows that a large n can help us bound the error of the selectivity estimate within a small range. The larger the data set is, the smaller the per record processing time cost will be. Actually, this bound is not as tight as the one in the previous theorem because the bound is obtained based on the previous theorem using more inequalities. Although the relative error here is based on n^2, and the previous one is based on g_s, which can be significantly smaller than n^2, depending on the data set and the similarity threshold, this is just because the two theorems measure two different values: relative similarity self-join size and selectivity of similarity self-join. Besides, the term 2^d is actually a bound coming from $\sum_{k=s}^{d}\binom{d}{k}$, which can be also much smaller than 2^d depending on d and s.

5.2.5 Our SelfJoinPairCount Algorithm (Online Scenario)

In the second step of our offline SelfJoinPairCount algorithm, we find the self-join size of the super-value streams by storing them either in memory or on disks. In this sub-section we discuss how to obtain an approximate self-join size more efficiently when the application requires a faster result while the super-value streams are too large to fit in memory. Furthermore, we study the performance changes after applying the faster but not precise self-join size estimates in our SelfJoinPairCount algorithm.

The need for faster self-join size estimates. When the data set is large and the generated super-value stream does not fit in memory, the offline SelfJoinPairCount algorithm stores the the super-value stream on disks. After processing the whole data set, SelfJoinPairCount sorts the super-value streams in a few passes to find their self-join sizes. Note that SelfJoinPairCounts generates up to $O(r2^d)$ super-values for each record; depending on the value of r, the size of super-value streams can be larger than the size of the original data set. For certain extremely large data sets or online data streams, even a few passes on the super-value streams can be too expensive or infeasible. It would be desired to process the data set in one pass and generate the result in memory promptly. The key challenge is how to find out the self-join sizes of the super-value streams efficiently in memory. Fortunately, there are existing algorithms efficiently approximating the self-join size of data streams in one pass using only a small amount of memory.

Estimating the self-join size of a data stream. A few sketching techniques can be used to estimate the self-join size of a data stream. Fast-AGMS or CountSketch, initially proposed by Charikar et al. [31] and used for self-join size estimation by Cormode and Garofalakis [46] is a more recent one. It has certain nice properties and theoretical guarantees, and we will use this sketching technique in our work. Note that Fast-AGMS is not the only option we have, other sketching techniques are also applicable to our algorithm. More sketching techniques are discussed in the related work section, and the previous chapter. For the ease of reading, we list the following result for Fast-AGMS; the proof can be found in the related work.

Lemma 5. *[55] Fast-AGMS gives an unbiased self-join size estimate for a data stream; the standard deviation of the estimate is at most*

$$\sqrt{\frac{4}{w}\sum_{\forall a<b} f_a^2 f_b^2},$$

where f_a and f_b are the frequencies of two different items, a and b.

Online SelfJoinPairCount. Our Online SelfJoinPairCount algorithm is similar to the offline one described in the previous sub-section. The difference is online SelfJoinPairCount uses sketching techniques to approximate the self-join sizes of the super-value streams, while offline SelfJoinPairCount finds the precise self-join sizes by storing the streams exactly, either in memory or on disks. The next theorem shows the performance of our online SelfJoinPair-

Count.

Theorem 9. *(Unbiased estimate and standard deviation from online SelfJoinPairCount) The online SelfJoinPairCount algorithm gives an unbiased estimate for the s-similarity self-join size, i.e.* $E[G_s] = g_s$*, and the standard deviation of* $\frac{G_s}{g_s}$ *is at most*

$$\binom{d}{s}\sqrt{\frac{1}{r}\binom{2(d-s)}{d-s}((1+\frac{2}{w})/g_s+\frac{2}{w}(1+\frac{n}{rg_s})^2)},$$

where w is the Fast-AGMS sketch width, d is the number of record attributes, s is the given similarity threshold, r *is the sampling factor,* g_s *is the true value of the similarity self-join size, and* G_s *is the estimated value.*

Proof. Since both offline SelfJoinPairCount and Fast-AGMS provides unbiased estimates, it is not hard to see the estimates from online SelfJoinPairCount are also unbiased.

Let Y'_k denote the variance of the self-join size estimate of the super-value stream using the Fast-AGMS algorithm, and Y_k denote the variance of the self-join size estimate using the offline SelfJoinPairCount algorithm as before, according to the law of total variance, we have

$$\begin{aligned} VAR[Y'_k] =& E[VAR[Y'_k|Y_k]] + VAR[E[Y'_k|Y_k]] \\ \le& E[\frac{2}{w}Y_k^2] + VAR[Y_k] \\ =& (1+\frac{2}{w})VAR[Y_k] + \frac{2}{w}E[Y_k]^2. \end{aligned}$$

Similar to the proof of Theorem 7, we have

$$
\begin{aligned}
VAR[G_s] &= VAR[\sum_{k=s}^{d} X_k] \le \frac{1}{r^4}\sum_{k=s}^{d}\binom{k}{s}^2 VAR[Y'_k] \\
&\le \frac{1}{r^4}(1+\frac{2}{w})\sum_{k=s}^{d}\binom{k}{s}^2 VAR[Y_k] + \frac{1}{r^4}\frac{2}{w}\sum_{k=s}^{d}\binom{k}{s}^2 E[Y_k]^2 \\
&\le (1+\frac{2}{w})\frac{1}{r}\binom{d}{s}^2\binom{2(d-s)}{d-s} g_s \\
&\quad + \frac{1}{r^4}\frac{2}{w}\sum_{k=s}^{d}\binom{k}{s}^2 (r^2\sum_{j=k}^{d}\binom{j}{k} x_j + nr\binom{d}{k})^2 \\
&\le (1+\frac{2}{w})\frac{1}{r}\binom{d}{s}^2\binom{2(d-s)}{d-s} g_s \\
&\quad + \frac{1}{r^4}\frac{2}{w}\sum_{k=s}^{d}\binom{k}{s}^2\binom{d}{k}^2 (r^2 g_s + nr)^2 \\
&= (1+\frac{2}{w})\frac{1}{r}\binom{d}{s}^2\binom{2(d-s)}{d-s} g_s \\
&\quad + \frac{1}{r^2}\frac{2}{w}\sum_{k=s}^{d}\binom{d}{s}^2\binom{d-s}{k-s}^2 (rg_s + n)^2 \\
&= (1+\frac{2}{w})\frac{1}{r}\binom{d}{s}^2\binom{2(d-s)}{d-s} g_s \\
&\quad + \frac{1}{r^2}\frac{2}{w}\binom{d}{s}^2\binom{2(d-s)}{d-s}(rg_s + n)^2.
\end{aligned}
$$

Therefore, the claim of the standard deviation of $\frac{G_s}{g_s}$ is true. □

Remarks. This theorem shows that as long as n is not much larger than rg_s, and w is reasonably large, the standard deviation of the online SelfJoinPairCount algorithm is very close to the offline one. Furthermore, to bound the standard deviation to a certain threshold, the value of w does not have to increase when n increases as long as g_s increases proportionally.

Theorem 10. *(Space and time cost to bound the selectivity estimation error) The online SelfJoinPairCount algorithm guarantees that the estimated selectivity of the similarity self-join deviates from the true value by at most*

ϵ with probability at least $1 - \lambda$, more precisely, $Pr[|\hat{\theta}_s - \theta_s| \leq \epsilon] \geq 1 - \lambda$, where $\hat{\theta}_s$ is the estimated selectivity and θ_s is the true value, and the total space cost is $O(log(1/\lambda)dw)$, and the time for processing each record is $O(log(1/\lambda)2^d\binom{d}{s}^2\binom{2(d-s)}{d-s}/(\epsilon^2 w))$.

Proof. Similar to the proof in Theorem 8, we have

$$\begin{aligned}
&Pr[|G_s - g_s| > \epsilon n^2] \leq \frac{VAR[G_s]}{\epsilon^2 n^4}\\
\leq &\frac{1}{\epsilon^2 n^2 r}\binom{d}{s}^2\binom{2(d-s)}{d-s}((1+\frac{2}{w}) + \frac{2}{w}(\frac{g_s}{n}+\frac{1}{r})^2)\\
\leq &\frac{1}{\epsilon^2 r}\binom{d}{s}^2\binom{2(d-s)}{d-s}(\frac{1}{n^2}(1+\frac{2}{w}) + \frac{2}{w}(1+\frac{1}{nr})^2).
\end{aligned}$$

Again, let the above term be $\frac{1}{8}$, then we have

$$\begin{aligned}
r \geq &\frac{8}{\epsilon^2}\binom{d}{s}^2\binom{2(d-s)}{d-s}(\frac{2}{n^2}(1+\frac{2}{w}) + \frac{8}{w})\\
\approx &\frac{64}{\epsilon^2 w}\binom{d}{s}^2\binom{2(d-s)}{d-s},
\end{aligned}$$

assuming $nr \geq 1$. Again, similar to the proof in Theorem 8, online Pair-Count repeat the same process $2log(1/\lambda)$ times to increase the success probability. □

Although r does not appear in the theorem, it is implicitly determined by other parameters. For example, a larger w would allow a smaller r while providing the same accuracy and confidence. Again, this bound is not as tight as the previous one, but it provides a bound that does not depend on the unknown statistics of the input data.

5.3 Result Set Size Estimation for Similarity Join and Search

In the previous section, we discuss our solutions for the similarity self-join size problem. Next, we will study how to estimate the result set size of other queries: similar join and search. Our solutions for these problems are extensions of the SelfJoinPairCount algorithm presented in Section 5.2.4.

5.3.1 Similarity Join Size Estimation

Definition 5. *(Join size). For two streams with N_1 and N_2 items, the join size is the number of exact matches among all $N_1 N_2$ item pairs. An item pair consists of items from different streams. Formally, the join size $F = \sum_{\forall i} f1_i f2_i$, where $f1_i$ and $f2_i$ are the frequency of a distinct item i in the first and the second stream.*

Definition 6. *(Similarity join size and selectivity). For two sets with n_1 and n_2 d-dimensional vectors, the s-similarity join size is the number of vector pairs that are at least s-similar; the selectivity of s-similarity join is the fraction of vector pairs that are at least s-similar among all possible $n_1 n_2$ pairs. A vector pair consists of two vectors from different data sets.*

Problem 2. *(Similarity join size estimation). Given two sets, with n_1 and n_2 d-dimensional vectors, each coordinate of the vectors is a non-negative value between 0 and $m - 1$, our goal is to efficiently estimate the size and the selectivity of an s-similarity join, where s is a given parameter. The similarity metric is the same as the one we used in defining the similarity self-join. The problem can be also defined similarly for two relations with n_1 and n_2 records and d attributes. Again, we do not distinguish the two cases in this chapter.*

5.3.2 Our JoinPairCount Algorithm

The main idea of our algorithm for solving the similarity join size estimation is the same the one for similarity self-join size estimation. That is, if a pair of records are similar, they should have a good chance to have a matching super-value when we randomly pick some attribute values from both records, and thus similar pairs are expected to contribute certain numbers to the super-value stream. Next, we will describe our JoinPairCount algorithms solving the similarity join size problem.

Processing the data sets. In the first step, JoinPairCount processes the two data sets independently, using the same algorithm as SelfJoinPairCount uses. The parameters for processing the data sets are the also same. This process generates two sets of super-value streams, one for each input data set. Each stream set has $(d-s+1)$-super-value streams. The streams can be either stored exactly as a list or approximately using sketches, as we discussed in the SelfJoinPairCount algorithm.

Finding the join size of the two data streams in the offline scenario. In this step, JoinPairCount finds the join sizes of the k-super-value stream pairs, $k \in [s, d]$. In the offline scenario, obtaining the join size for two data streams can be done as follows. If one of the streams, let us call it the first stream, fits in memory [6] , a hash index for this stream can be created to speed up the process of detecting duplicates and counting the frequencies of each distinct item . Accordingly, the join size can be computed by checking the corresponding frequency in the second stream for each item in the first stream. Having a proper size hash index, this process can be done in constant time for each checking. Thus, the time cost for finding the join size is linear in the longer stream size. If neither of the two streams fit in memory, one can sort both streams on disks. External sorting needs $O(log^{R}_{M-1})$ passes to sort a data stream, where M is the number of memory pages available, and R is the number of blocks occupied by the data stream. Having two sorted streams, computing the join size is straightforward.

---

[6] This can be known before processing the data set provided that the input data set size is known, since the number of super-values each record generates is known.

Estimating the similarity join size. After finding the $d-s+1$ join sizes of the super-value stream pairs, JoinPairCount computes the s-similarity join size in the same way as in the similarity self-join estimation. Let Y_k denote the join size of the k-super-value stream pair, the number of k-similar record pairs among the two input data sets, denoted by x_k can be estimated using Eq. 5.1. Similarly, X_k denotes the estimated value of x_k. as defined in Section 5.2.4.

Algorithm analysis. Since most of the analysis is similar to that of SelfJoinPairCount, we only list the result, and omit the proof.

Corollary 6. *(Unbiased similarity join size estimate and standard deviation of offline JoinPairCount) The offline JoinPairCount algorithm gives an unbiased estimate of the s-similarity join size, i.e.* $E[G_s] = g_s$, *and the standard deviation of* $\frac{G_s}{g_s}$ *is at most* $\binom{d}{s}\sqrt{\frac{1}{r}\binom{2(d-s)}{d-s}}/g_s$, *where* G_s *is the estimated similarity join size, and* g_s *is the true value.*

Corollary 7. *(Space and time cost) The offline JoinPairCount algorithm guarantees that the estimated selectivity of the similarity join deviates from the true value by at most* ϵ *with probability at least* $1-\lambda$. *More precisely,* $Pr[|\hat{\theta}_s - \theta_s| \leq \epsilon] \geq 1-\lambda$, *where* $\hat{\theta}_s$ *is the estimated selectivity and* θ_s *is the true value. The space cost for processing the data sets is* $O(log(1/\lambda)\binom{d}{s}^2\binom{2(d-s)}{d-s}2^d(n_1 + n_2)/(\epsilon^2 n_1 n_2))$, *and the time for processing each record is* $O(log(1/\lambda)\binom{d}{s}^2\binom{2(d-s)}{d-s}2^d/(\epsilon^2 n_1 n_2))$; *where* n_1 *and* n_2 *are the number of records in the two input data sets, d is the number of attributes, and s is the given similarity threshold.*

Remarks. It is not hard to see that these two claims are very similar to the ones for SelfJoinPairCount. The only difference is the data set sizes. In the statements above, G_s and g_s represent different values. Also, n_1 and n_2 replace n. But the analysis is still similar to those for SelfJoinPairCount, so we do not repeat them here.

Finding the join size of the two data streams in the online scenario. Similar to the discussion in Section 5.2.5, finding the join sizes of large data stream pairs can be costly for certain applications. In this case, online

JoinPairCount can also use sketching techniques to approximate the join size efficiently. Fast-AGMS sketches can also be used to efficiently approximate join sizes. The sketch construction is the same for each of the two streams, using the same sets of hash functions and same sketch widths, but the process of generating the results is slightly different: JoinPairCount sequentially checks each pair of corresponding counters of the two sketches, one for each data set, and computes the sum of the products of all counter value pairs.

Corollary 8. *(Unbiased similarity join estimate and standard deviation from online JoinPairCount) The online JoinPairCount algorithm gives an unbiased estimate for the s-similarity join size, i.e.* $E[G_s] = g_s$*, and the standard deviation of* $\frac{G_s}{g_s}$ *is at most*

$$\binom{d}{s}\sqrt{\frac{1}{r}\binom{2(d-s)}{d-s}((1+\frac{2}{w})/g_s + \frac{2}{w}(1+\frac{n_1+n_2}{rg_s})^2)},$$

where w *is the Fast-AGMS sketch width,* d *is the number of attributes of records,* s *is the given similarity threshold,* r *is the sampling rate,* g_s *is the true value of the similarity join size, and* G_s *is the estimated value.*

Corollary 9. *(Space and time cost for bounding the selectivity estimation error) The online JoinPairCount algorithm guarantees that the estimated selectivity of the similarity join deviates from the true value by at most* ϵ *with probability at least* $1-\lambda$*, more precisely,* $Pr[|\hat{\theta}_s - \theta_s| \leq \epsilon] \geq 1-\lambda$*, where* $\hat{\theta}_s$ *is the estimated selectivity and* θ_s *is the true value, and the total space cost is* $O(log(1/\lambda)dw)$*, and the time for processing each record is* $O(log(1/\lambda)2^d\binom{d}{s}^2\binom{2(d-s)}{d-s}/(\epsilon^2 w))$.

5.3.3 Result Set Size Estimation for Similarity Search

In the previous section, we discuss our solutions for the similarity join size problem. Next, we will show that our PairCount algorithm can also be applied to the selectivity estimation for similarity search.

Definition 7. *(Similarity search, result set size and selectivity). Given a set* *of n d-dimensional vectors and a query vector, a similarity search is a query for finding the vectors that are at least s-similar to the given vector in the given data set. The result set size is the number of vectors in the query result. The selectivity is the ratio between result set size and the data set size. These terms can be defined similarly for relations and records.*

Problem 3. *(Result set size and selectivity estimation for similarity searches). Given a set of n d-dimensional vectors and a query vector, the goal is to efficiently estimate the result set size and selectivity of an s-similarity search.*

Our offline SearchPairCount solution. In fact, a similarity search query can be considered as a special similarity join query, where one side of the join is the given relation, and the other side is another special relation with only one record, which is the given query record. Therefore, the JoinPairCount algorithm can be directly applied to estimate the selectivity of a similarity search. The results are listed below, and the proofs are omitted since they can be directly adapted from the those of JoinPairCount.

Corollary 10. *(Unbiased estimate and standard deviation of offline SearchPairCount) The offline SearchPairCount algorithm gives an unbiased estimate for an s-similarity search, i.e. $E[G_s] = g_s$, and the standard deviation of $\frac{G_s}{g_s}$ is at most $\binom{d}{s}\sqrt{\frac{1}{r}\binom{2(d-s)}{d-s}}/g_s$, where G_s is the estimated result set size of the similarity search, and g_s is the true value.*

Corollary 11. *(Space and time cost) The offline SearchPairCount algorithm guarantees that the estimated selectivity of the similarity search deviates from the true value by at most ϵ with probability at least $1 - \lambda$. More precisely, $Pr[|\hat{\theta}_s - \theta_s| \leq \epsilon] \geq 1 - \lambda$, where $\hat{\theta}_s$ is the estimated selectivity and θ_s is the true value. The space cost for processing the data sets is $O(log(1/\lambda)\binom{d}{s}^2\binom{2(d-s)}{d-s}2^d/\epsilon^2)$, and the time for processing the query is $O(log(1/\lambda)\binom{d}{s}^2\binom{2(d-s)}{d-s}2^d/\epsilon^2 n)$, where d is the number of attributes, n is the number of records in the given data set, and s is the given similarity threshold.*

Our online SearchPairCount Solution. Similar to the self-join and join query, we also provide an online version of our SearchPairCount algorithm, which applies Fast-AGMS to estimate the frequency of a given item in a data stream.

Corollary 12. *(Unbiased estimate and standard deviation of online SearchPairCount) The online SearchPairCount algorithm gives an unbiased estimate for the s-similarity size, i.e.* $E[G_s] = g_s$*, and the standard deviation of* $\frac{G_s}{g_s}$ *is at most*

$$\binom{d}{s}\sqrt{\frac{1}{r}\binom{2(d-s)}{d-s}((1+\frac{2}{w})/g_s+\frac{2}{w}(1+\frac{n}{rg_s})^2)},$$

where w *is the Fast-AGMS sketch width,* n *is the number of records in the given data set,* d *is the number of attributes of records,* s *is the given similarity threshold,* r *is the sampling rate,* g_s *is the true value of the result set size of the similarity search, and* G_s *is the estimated value.*

Corollary 13. *(Space and time cost) The online SearchPairCount algorithm guarantees that the estimated selectivity of the similarity search deviates from the true value by at most* ϵ *with probability at least* $1-\lambda$*, more precisely,* $Pr[|\hat{\theta}_s - \theta_s| \leq \epsilon] \geq 1-\lambda$*, where* $\hat{\theta}_s$ *is the estimated selectivity and* θ_s *is the true value, and the total space cost is* $O(log(1/\lambda)dw)$*, and the time for processing each record is* $O(log(1/\lambda)2^d\binom{d}{s}^2\binom{2(d-s)}{d-s}/(\epsilon^2 w))$.

5.4 Extensions and Applications

From vector similarity to set similarity In the previous sections we have mentioned that min-wise independent hashing can be used to convert sets into vectors such that the s-similarity of a vector pair is expected to give the Jaccard similarity of two sets. In addition to min-wise independent hashing, sets can also be converted into vectors using random projection such that the s-similarity of a vector pair is expected to give a function of the Cosine similarity of the original pair of sets.

In addition to Jaccard and Cosine similarities, our PairCount algorithm actually can be adapted to handle set similarities provided that the users are interested in the number of common items between two item sets. Assuming the maximum number of items among all sets, d, is known, the algorithm processing input data sets can be adapted as follows: for each item sets with less than d items, the algorithm pads the item set with distinct items such that each item set has d items. For example, strings consisting of the item set number and distinct indices in the set can be padded. Having the padded item sets, PairCount can process each item set as a vector.

PairCount in parallel. One nice property of our PairCount algorithms is that they can be easily used in distributed environments, which is important for processing large data sets. One can cut a huge data set or a data stream into many smaller pieces and distribute them to multiple machines. Each machine can run the PairCount algorithm using the same parameters, generating multiple super-value streams or sketches. After processing the whole data set, each machine can sort much smaller super-value streams if needed, and sent the processed super-value streams or sketches to a centralized server and let the server compute the final estimates. In fact, in many data intensive applications such as Web crawling, the web pages are collected in a distributed fashion; our PairCount algorithms can be readily applied in these cases.

Similarity query optimization. As similarity queries becoming more popular and important, DBMS needs to consider optimizing the similarity queries, especially similarity joins. One important statistic for helping optimizer pick a better query plan is the selectivity of each intermediate step. Using the online PairCount algorithms, those statistics can be efficiently estimated in one pass over the data.

Similarity query refinement. In the similarity search applications, it is normal that the users are not able to write a perfect query in the first time; they may issue a query generating a large result set. In this case, it would be desired if the system can efficiently estimate the result set size when receiving the query, and quickly let the users know when the result set is too large. Having sorted super-value streams or sketches stored on disk as an auxiliary data structure and using the SearchPairCount algorithm, the system can return an estimate efficiently.

5.5 Experiments

In this section, we will provide experimental results verifying the effectiveness and efficiency of our algorithms. Since the self-join size estimation is the core of our algorithms, we focus on the offline and online SelfJoinPairCount algorithm in this section.

5.5.1 Experimental Setup

Data sets. The data set we used was a set of paper titles in computer science obtained from DBLP. After some standard text preprocessing, such as removing stop words and stemming, we fingerprinted each paper title into 6 super-shingles using min-wise independent hashing to simulate the experimental settings in the works of Henzinger [77] and Broder et al. [18, 23], where their goal was to list all similar web document pairs. In both our and their experiments, each super-shingle is a 64-bit fingerprint. At the end, we had $467,468$ records, each with 6 attributes (super-shingles). Next, we report the experimental result of Offline SelfJoinPairCount.

Implementation details. All experiments were run on an AMD64/4000 Scientific Linux machine, with 2G memory, implemented in C. The Fast-AMGS implementation was adapted from MassDal [95].

5.5.2 Offline SelfJoinPairCount

First, we show the performance of our SelfJoinPairCount algorithm in an offline scenario, which means we store the super-value stream exactly rather than using a sketch summary.

Effectiveness of SelfJoinPairCount. In this experiment, we set the sample factor to 0.5, which means SelfJoinPairCount randomly picked 32 super-values. We ran our SelfJoinPairCount on the first $200K$ rows of the data set. The estimated number of k-similar pairs, the true number and the relative errors are listed in Table 5.1. These numbers are consistent with the

Table 5.1: SelfJoinPairCount relative errors (estimate-truth)/truth.

k	TrueValue	Estimate	Relative error
6	219356	219356	0
5	191310	188720	-1.35%
4	1690036	1701575	0.68%
3	14706402	14822275	0.79%
2	87385874	87504347	0.14%
1	417430350	417105860	-0.08%
0	39478376672	39478457866	0.0002%

Table 5.2: SelfJoinPairCount relative errors with different data set sizes.

Data set size	TrueValue	Relative error
25K	46970	8.76%
50K	126762	5.11%
100K	473408	1.3%
200K	1690036	0.68%
400K	6117538	-1.15%

result in Theorem 7 and show the effectiveness of SelfJoinPairCount. Note that when $k = 6$, the result was actually the self-join size of the 6-super-value stream. Thus, there was no error in it.

Varying the data set size. To see the scalability of our algorithm, we varied the size of the data sets from $10K$ to $400K$ records, still with sampling factor 0.5. The results are shown in Table 5.2 for the number of record pairs that are 4-similar. From the table we can see that in general, the relative error drops when the data set is larger, and the true value is also larger.

Varying the sampling factor. In this set of experiments we varied the sampling factor while fixing the data set size to 100K. The relative errors are shown in Table 5.3. The results show that the error rate decreases when the sampling factor increases in general.

Table 5.3: SelfJoinPairCount relative errors with different sampling factors.

Sampling factor	k=5	k=4	k=3
0.75	-3.89%	2.47%	-1.08%
0.5	-2.6%	1.3%	1.3 %
0.25	6.97%	3.93%	-2.20%

Table 5.4: SelfJoinPairCount relative errors (estimate-truth)/truth with different dimensionality.

d	TrueValue	Estimate	Relative error
8	44980	47704	6.06%
7	17756.67	18480	-3.91%
6	54442	53028	-2.60%
5	239282	240552.5	0.53%
4	1273288	1286976	1.08%

Varying the number of attributes. In this set of experiments, we fixed the query to finding the number of pairs that are $(0.8d)$-similar, and increased the number of attributes d. The data sets were generated by using different number of min-hashes for each record in the min-wise independent hashing process. There were $100K$ records in each data set. The sampling factors were set to 0.5. The results listed in Table 5.4 shows that the relative error increased exponentially with d. Therefore, JoinPairCount may not work well in high-dimensional cases.

5.5.3 Online SelfJoinPairCount

The previous experiments show the performance of offline SelfJoinPairCount; next we show how the estimation accuracy changes by using the Fast-AGMS sketches to find the self-join size of super-value streams with only a very small amount of space.

Effectiveness of SelfJoinPairCount. Similar to the offline SelfJoinPairCount experiments, we set the sample factor to 0.5, and ran our SelfJoinPairCount on the first $200K$ rows of the data set. The sketch width (number of counters) was set to 1000, and the sketch depth was 3. The estimated number of k-similar pairs, the true numbers and the relative errors are listed in Table 5.5. The results were less accurate than those from offline estimates, but considering the small amount of space costs and the online property, we consider them reasonably good.

Varying the data set size. Next, we report the results of online Self-

Table 5.5: Online SelfJoinPairCount relative errors (estimate-truth)/truth.

k	TrueValue	Estimate	PairCount error
6	219356	220866	0.69%
5	191310	223620	16.89%
4	1690036	1405057	-16.86%
3	14706402	16455032	11.89%
2	87385874	81663188	-6.55%
1	417430350	389797943	-6.62%
0	39478376672	39510234292	0.08%

Table 5.6: Online SelfJoinPairCount relative errors with different data set sizes.

Data set size	TrueValue	Relative error
100K	473408	47.77%
200K	1690036	-16.86%
300K	2557746	-27.56%
400K	6117538	-2.94 %

JoinPairCount on data sets of different sizes while keeping other parameters the same. The results are shown in Table 5.6 for the number of record pairs that are 4-similar. In general, the error rates decreased the input size was increased. except for the $50K$ and $100K$ data sets.

Comparison with random sampling. In this set of experiments, we compared our online SelfJoinPairCount with random sampling by keeping their space costs the same. As before, we set the similarity threshold to 4, sampling factor to 0.5, and sketch width and depth to 1000 and 3 respectively. Accordingly, our online SelfJoinPairCount used 9000 counters in total. Since each input data record is 6 64-bit fingerprints, assuming each counter needs 32 bits, we set the sample size to 750. i.e. 750 records in the sample. We ran both algorithms on data sets with different sizes: the first $200K$, $300K$ and $400K$ records. The results reported in Table 5.7 show that our online SelfJoinPairCount is significantly more accurate than random sampling.

In terms of time comparison, we consider two stages: data summarization stage and query answering stage. At the data summarization stage, random sampling took R operations; online SelfJoinPairCount took $nhr\sum_{k=s}^{d}\binom{d}{k}$ operations, where R is the sample size, and h is the sketch depth. In our ex-

Table 5.7: Online SelfJoinPairCount and random sample relative errors.

k	TrueValue	Random sample error	PairCount error
	200K data		
6	219356	56.00%	0.69%
5	191310	-25.47%	16.89%
4	1690036	-66.25%	-16.86%
	300K data		
6	336668	-10.89%	1.28 %
5	380318	68.28%	34.06%
4	3530756	63.14%	-27.56%
	400K data		
6	457466	-12.56%	1.59 %
5	618036	83.87%	2.91%
4	6117538	-53.56%	-2.94%

perimental settings, random sampling is clearly faster at this stage. However, the difference will become smaller when the data set is larger, since the time complexity of SelfJoinPairCount is linear in the data set size, while random sampling is quadratic in the sample size and is expected to become slow with large sample sizes.

At the second stage, having the data summary, our online SelfJoinPairCount needed $(d-s+1)wh+(d-s+1)$ operations, which basically is the time for scanning the data summary once, while random sampling needed $S(S-1)/2$ operations. Thus, online SelfJoinPairCount is much faster at this stage. This is an important advantage for our method when the data summary can be constructed in advance before queries arrive.

5.6 Related Work

This work is related to multiple areas including similarity search, streaming algorithms, selectivity estimation for different types of queries, data cleaning, fuzzy/near duplicate detection and set/text join.

Similarity search and locality sensitive hashing. Our algorithms are closely related to locality sensitive hashing (LSH) [73, 80], where the main idea is: two multidimensional points that are "close" have higher chance to be mapped to the same hash value than those points that are "far" from each other using LSH. "Closeness" is measured by different distance functions such as Hamming and Euclidean distances. For LSH, there are different types of hash functions. Charikar [29] considers min-wise independent hashing [21, 20] as a special LSH, and also proposes new LSHs, one of which can map sets into vectors such that the cosine similarity between sets can be estimated by checking the number of vector coordinates in common. Although related, our work is different from LSH since our algorithms use sampling rather than hashing.

Streaming algorithms and sketching technique. In their seminal work [4], Alon et al. propose AMS sketching techniques to estimate self-join sizes of data streams using a small amount of space. This sketching technique has been widely extended and applied. For example, it can be used to answer different types of frequency related queries on data streams such as join size estimation [3] and frequent item tracking [31]. Another popular data structure that can be used for data streams is Bloom filters [17]. Deng and Rafiei extend Bloom filters to detect exact duplicates in a streaming environment. Cormode and Muthukrishnan [48] show that Bloom filters can be extended to answer different types of frequency related queries such as join/self-join size estimation and item frequency query; they call their data structure Count-min. Deng and Rafiei [55] further extend Count-min and show that Count-min can perform as well as Fast-AGMS [31], an extension of AMS sketch that is suitable for data streams, while Count-min provides more flexible and powerful functionalities.

Selectivity estimation. This topic has been widely studied in different scenarios such as substring queries [39, 33], relational queries [70, 109], Containment Join [117], and spatial join [53, 65]. Their query types or similarity measures are different from those in our work. Jin and Li [83] studied the selectivity estimation problem of fuzzy string queries where the similarity is measured by edit distance. They also show that their work can be extended to Jaccard similarity, but they only consider selectivity estimation for fuzzy string predicates while our work is applicable to selectivity estimations for both join/self-join and similarity search.

Data cleaning, fuzzy/near duplicate detection and set/text join. Another active research area related to our work is data cleaning [32, 36, 34, 7, 77, 18]. Elmagarmid et al. [60] recently give a survey of the work on the fuzzy/near duplicate detection problem. More work related to this topic can be found there. As discussed in the introduction section, one important goal of data cleaning is finding all near/fuzzy duplicate pairs based on certain similarity measures. Although different prunings may improve the efficiency, the nature of this problem determines that it can be a very expensive operation for large data set even if the pruning is perfect, since the true answer set can increase quadratically with the data set size, which motivates our work.

5.7 Summary and Possible Extension

In this chapter, we present a set of algorithms converting or summarizing multidimensional vectors or records such that frequency related similarity queries can be efficiently answered. Specifically, our PairCount algorithms focus on estimating the result set sizes of similarity self-join, join and search queries on multidimensional vectors, and give efficient and effective approximate answers. The main idea of our algorithms are mapping multidimensional vectors into multiple one-dimensional data streams; this transforms the result set size estimation problem for multidimensional vetoes to frequency related query estimation problem for data streams, where efficient sketching techniques can be applied. The effectiveness and efficiency of the algorithms are theoretically analyzed and experimentally evaluated. Several extensions and applications are also described.

Our current similarity measures are mainly based on Hamming distance. Extending the main idea of this work to other metrics such as Euclidean distance is being considered. Also, future research may study the connection between our work and containment and spatial join.

Chapter 6

Conclusions and Future Work

This thesis studies the problem of efficiently answering frequency related queries on streaming data using a small amount of space. The queries include membership query, frequency query, join/self-join size estimation and their similarity versions. The algorithms proposed in this thesis are extended from or based on well-known sketching or sampling techniques. Both theoretical and experimental evidence show that the proposed methods improve upon the-state-of-the-art.

Direct extensions of our work have been discussed in the previous chapters. Here we present several related directions in a broad sense for future research.

Efficiently finding certain statistics for a large data set can be very useful in many different scenarios. For example, efficiently approximating pairwise distance distributions of a large data set may be helpful in selecting an indexing strategy or a better clustering algorithm, or even showing that some clustering/indexing algorithms may not work at all. Also, in detecting outliers, quickly estimating the number of outliers under different outlier definitions can be helpful to users when they are formulating their queries. One may define an outlier as a data point that has no neighbors within a certain distance threshold; others may define an outlier as a data point that has less than 5 neighbors. Knowing the result set size in advance may help user reformulating their queries before starting a relatively expensive outlier detection. However, efficiently and accurately estimating aforementioned statistics may not be easy.

A more general research direction is: given an expensive operation on a large data set, find out if knowing certain statistics of the data set can help to improve the efficiency of the expensive operation. If yes, then how can we obtain the statistics efficiently, possibly using approximation algorithms.

We may also want to estimate statistics of large data sets, not for the sake of a particular task, but for exploratory purposes and to enlarge our knowledge about the world.

Bibliography

[1] D. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. The design of the borealis stream processing engine. In *First Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 277–289, 2005.

[2] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[3] N. Alon, P. B. Gibbons, Y. Matias, and M. Szegedy. Tracking join and self-join sizes in limited storage. In *PODS*, 1999.

[4] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *JCSS*, 58(1), 1999. Also in: *STOC*, 1996.

[5] R. Ananthakrishna, S. Chaudhuri, and V. Ganti. Eliminating fuzzy duplicates in data warehouses. In *Proc. of VLDB*, 2002.

[6] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom. Stream: The stanford stream data manager. *IEEE Data Eng. Bull.*, 26(1):19–26, 2003.

[7] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *Proc. of International Conference on Very Large Data Bases (VLDB)*, pages 918–929, 2006.

[8] Internet Archive. `http://www.archive.org/`.

[9] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. of PODS*, 2002.

[10] B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. In *SODA*, pages 633–634, 2002.

[11] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *Proc. of ICDE*, 2004.

[12] F. Baboescu, S. Singh, and G. Varghese. Packet classification for core routers: Is there an alternative to cams? In *Proc. of INFOCOMM*, 2003.

[13] H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. Zdonik. Retrospective on aurora. *The VLDB Journal*, 13(4):370–383, 2004.

[14] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *RANDOM*, pages 1–10, 2002.

[15] D. Barbará, W. DuMouchel, C. Faloutsos, P. J. Haas, J. M. Hellerstein, Y. E. Ioannidis, H. V. Jagadish, T. Johnson, R. T. Ng, V. Poosala, K. A. Ross, and K. C. Sevcik. The new jersey data reduction report. *IEEE Data Eng. Bull.*, 20(4):3–45, 1997.

[16] M. Bilenko and R. J. Mooney. Adaptive duplicate detection using learnable string similarity m easures. In *Proc. of KDD*, 2003.

[17] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13(7), 1970.

[18] A. Z. Broder. On the resemblance and containment of documents. In *Proc. of Compression and Complexity of Sequences*, 1997.

[19] A. Z. Broder. Identifying and filtering near-duplicate documents. In *Proc. of Combinatorial Pattern Matching (CPM)*, pages 1–10, 2000.

[20] A. Z. Broder. Min-wise independent permutations: Theory and practice. In *Proc. of the International Colloquium on Automata, Languages and Programming (ICALP)*, page 808, 2000.

[21] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences (JCSS)*, 60(3), 2000. Also in: *Proc. of the ACM Symposium on Theory of Computing (STOC)*, 1998.

[22] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. *Computer Networks*, 29(8-13):1157–1166, 1997.

[23] A. Z. Broder, S. C. Glassman, M. S. Manasse, and Geoffrey Zweig. Syntactic clustering of the web. *Computer Networks*, 29(8-13):1157–1166, 1997.

[24] A. Z. Broder, M. Najork, and J. L. Wiener. Efficient url caching for world wide web crawling. In *Proc. of WWW*, 2003.

[25] N. Brownlee, C. Mills, and G. Ruth. Traffic flow measurement: Architecture. *RFC 2722*, October 1999.

[26] R. Caceres, N. G. Duffield, A. Feldmann, J. Friedmann, A. Greenberg, R. Greer, T. Johnson, C. Kalmanek, B. Krishnamurthy, D. Lavelle, P. Mishra, K. K. Ramakrishnan, J. Rexford, F. True, and J. E. van der Merwe. Measurement and analysis of IP network usage and behavior. *IEEE Communications*, 38(5):144–151, 2000.

[27] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. B.Zdonik. Monitoring streams - a new class of data management applications. In *Proc. of VLDB*, 2002.

[28] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *Proc. of CIDR*, 2003.

[29] M. Charikar. Similarity estimation techniques from rounding algorithms. In *Proc. of the ACM Symposium on Theory of Computing (STOC)*, pages 380–388, 2002.

[30] M. Charikar, S. Chaudhuri, R. Motwani, and V. R. Narasayya. Towards estimation error guarantees for distinct values. In *PODS*, pages 268–279, 2000.

[31] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. *Theoretical Computer Science (TCS)*, 312(1), 2004. Also in: *ICALP*, 2002.

[32] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 313–324, 2003.

[33] S. Chaudhuri, V. Ganti, and L. Gravano. Selectivity estimation for string predicates: Overcoming the underestimation problem. In *Proc. of the International Conference on Data Engineering (ICDE)*, pages 227–238, 2004.

[34] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *Proc. of the International Conference on Data Engineering (ICDE)*, page 5, 2006.

[35] S. Chaudhuri, V. Ganti, and R. Motwani. Robust identification of fuzzy duplicates. In *Proc. of ICDE*, 2005.

[36] S. Chaudhuri, V. Ganti, and R. Motwani. Robust identification of fuzzy duplicates. In *Proc. of the International Conference on Data Engineering (ICDE)*, pages 865–876, 2005.

[37] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *Proc. of SIGMOD*, 2000.

[38] Y. Chen, G. Dong, J. Han, B. W. Wah, and J. Wang. Multi-dimensional regression analysis of time-series data streams. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, pages 323–334, August 2002.

[39] Z. Chen, F. Korn, N. Koudas, and S. Muthukrishnan. Generalized substring selectivity estimation. *Journal of Computer and System Sciences (JCSS)*, 66(1):98–132, 2003. Also in: *PODS*, 2000.

[40] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. B. Zdonik. Scalable distributed stream processing. In *First Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003.

[41] E. Cohen and M. Strauss. Maintaining time-decaying stream aggregates. In *Proc. of PODS*, 2003.

[42] S. Cohen and Y. Matias. Spectral bloom filters. In *Proc. of SIGMOD*, 2003.

[43] R. B. Cooper. *Introduction to Queueing Theory(2nd edition)*. Elsevier, 1981.

[44] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 2nd Edition*. The MIT Press, 2001.

[45] G. Cormode, M. Datar, P. Indyk, and S.Muthukrishnan. Comparing data streams using hamming norms (how to zero in). *IEEE Trans. Knowl. Data Eng.*, 15(3):529–540, 2003.

[46] G. Cormode and M. N. Garofalakis. Sketching streams through the net: Distributed approximate query tracking. In *VLDB*, 2005.

[47] G. Cormode, T. Johnson, F. Korn, S. Muthukrishnan, O. Spatscheck, and D. Srivastava. Holistic udafs at streaming speeds. In *SIGMOD*, 2004.

[48] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1), 2005. Also in: *LATIN*, 2004.

[49] G. Cormode and S. Muthukrishnan. Summarizing and mining skewed data streams. In *SIAM International Conference on Data Mining (SDM)*, 2005.

[50] G. Cormode, S. Muthukrishnan, and I. Rozenbaum. Summarizing and mining inverse distributions on data streams via dynamic inverse sampling. In *VLDB*, pages 25–36, 2005.

[51] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *Proc. of SIGMOD*, 2003.

[52] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. The gigascope stream database. *IEEE Data Engineering Bulletin*, 26(1):27–32, 2003.

[53] A. Das, J. Gehrke, and M. Riedewald. Approximation techniques for spatial data. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 695–706, 2004.

[54] M. Datar and S. Muthukrishnan. Estimating rarity and similarity over data stream windows. In *ESA*, pages 323–334, 2002.

[55] F. Deng and D. Rafiei. New estimation algorithms for streaming data: Count-min can do more. `http://www.cs.ualberta.ca/~fandeng/paper/cmm.pdf`.

[56] F. Deng and D. Rafiei. Approximately detecting duplicates for streaming data using stable bloom filters. In *SIGMOD*, 2006.

[57] N. G. Duffield. Sampling for passive internet measurement: A review. *Statistical Science*, 19(3):472–498, 2004.

[58] N. G. Duffield, C. Lund, and M. Thorup. Estimating flow distributions from sampled flow statistics. *IEEE/ACM Trans. Netw.*, 13(5):933–946, 2005. also in:*SIGCOMM*, 2003.

[59] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering(TKDE)*, 19(1):1–16, 2007.

[60] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering(TKDE)*, 19(1):1–16, 2007.

[61] C. Estan, K. Keys, D. Moore, and G. Varghese. Building a better netflow. In *Proc. of SIGCOMM*, 2004.

[62] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *Proc. of SIGCOMM*, 2002.

[63] C. Estan and G. Varghese. Data streaming in computer networks. In *Proc. of Workshop on Management and Processing of Data Streams(MPDS) in cooperation with SIGMOD/PODS*, 2003.

[64] C. Estan, G. Varghese, and M. Fisk. Bitmap algorithms for counting active flows on high speed links. In *Internet Measurement Comference*, pages 153–166, 2003.

[65] C. Faloutsos, B. Seeger, A. J. M. Traina, and C. Traina Jr. Spatial join selectivity using power laws. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 177–188, 2000.

[66] L. Fan, P. Cao, J. M. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3), 2000. also in:*SIGCOMM*, 1998.

[67] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, 1985.

[68] S. Ganguly, M. N. Garofalakis, and R. Rastogi. Processing data-stream join aggregates using skimmed sketches. In *EDBT*, 2004.

[69] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database System Implementation.* Prentice Hall, 2000.

[70] L. Getoor, B. Taskar, and D. Koller. Selectivity estimation using probabilistic models. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 461–472, 2001.

[71] P. B. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *VLDB*, pages 541–550, 2001.

[72] P. B. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *SIGMOD Conference*, pages 331–342, 1998.

[73] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proc. of International Conference on Very Large Data Bases (VLDB)*, pages 518–529, 1999.

[74] M. Grossglauser and J. Rexford. Passive traffic measurement for ip operations. In *The Internet as a Large-Scale Complex System*, pages 91–120. Oxford University Press, 2005.

[75] P. J. Haas, J. F. Naughton, S. Seshadri, and L. Stokes. Sampling-based estimation of the number of distinct values of an attribute. In *VLDB*, 1995.

[76] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, 3rd Edition.* Morgan-Kaufmann, 2002.

[77] M. R. Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *SIGIR*, pages 284–291, 2006.

[78] A. Heydon and M. Najork. Mercator: A scalable, extensible web crawler. *World Wide Web*, 2(4), 1999.

[79] Cisco System Inc. Cisco network accounting services. `http://www.cisco.com/warp/public/cc/pd/iosw/prodlit/nwact_wp.pdf`, 2002.

[80] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proc. of the ACM Symposium on Theory of Computing (STOC)*, pages 604–613, 1998.

[81] P. Indyk and D. P. Woodruff. Optimal approximations of the frequency moments of data streams. In *STOC*, 2005.

[82] C. Jermaine, A. Pol, and S. Arumugam. Online maintenance of very large random samples. In *SIGMOD Conference*, pages 299–310, 2004.

[83] L. Jin and C. Li. Selectivity estimation for fuzzy string predicates in large data sets. In *Proc. of International Conference on Very Large Data Bases (VLDB)*, pages 397–408, 2005.

[84] R. M. Karp, S. Shenker, and C. H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *TODS*, 28(1), 2003.

[85] F. Korn, S. Muthukrishnan, and Y. Wu. Modeling skew in data streams. In *SIGMOD*, 2006.

[86] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen. Sketch-based change detection: methods, evaluation, and applications. In *Proc. of the Internet Measurement Conference(IMC)*, 2003.

[87] S. Krishnamurthy, S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Madden, F. Reiss, and M. A. Shah. Telegraphcq: An architectural status report. *IEEE Data Engineering Bulletin*, 26(1):11–18, 2003.

[88] A. Kumar, M. Sung, J. Xu, and J. Wang. Data streaming algorithms for efficient and accurate estimation of flow size distribution. In *SIGMETRICS*, pages 177–188, 2004.

[89] A. Kumar, J. Xu, J. Wang, O. Spatscheck, and L. Li. Space-code bloom filter for efficient per-flow traffic measurement. In *INFOCOM*, 2004.

[90] A. Lerner and D. Shasha. The virtues and challenges of ad hoc + streams querying in finance. *IEEE Data Engineering Bulletin*, 26(1):49–56, March 2003.

[91] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *Proc. of VLDB*, 1986.

[92] U. Manber and S. Wu. An algorithm for approximate membership checking with application to password security. *Information Processing Letters*, 50(4), 1994.

[93] A. Manjhi, V. Shkapenyuk, K. Dhamdhere, and C. Olston. Finding (recently) frequent items in distributed data streams. In *Proceedings of the 21st International Conference on Data Engineering*, pages 754–765, April 2005.

[94] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proc. of VLDB*, pages 346–357, September 2002.

[95] MassDal. Massdal public code bank. `http://www.cs.rutgers.edu/~muthu/massdal-code-index.html`, 2006.

[96] A. Metwally, D. Agrawal, and A. E. Abbadi. Duplicate detection in click streams. In *Proc. of WWW*, 2005.

[97] A. Metwally, D. Agrawal, and A. E. Abbadi. Efficient computation of frequent and top-k elements in data streams. In *ICDT*, 2005.

[98] M. Mitzenmacher. Compressed bloom filters. *IEEE/ACM Trans. Netw.*, 10(5):604–612, 2002.

[99] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[100] S. Muthukrishnan. Data streams: Algorithms and applications. In *http://athos.rutgers.edu/ muthu/stream-1-1.ps*, 2003.

[101] J. F. Naughton, D. J. DeWitt, D. Maier, A. Aboulnaga, J. Chen, L. Galanis, J. Kang, R. Krishnamurthy, Q. Luo, N. Prakash, R. Ramamurthy, J. Shanmugasundaram, F. Tian, K. Tufte, S. Viglas, Y. Wang, C. Zhang, B. Jackson, A. Gupta, and R. Chen. The niagara internet query system. *IEEE Data Engineering Bulletin*, 24(2):27–33, 2001.

[102] F. Olken. Random sampling from databases. Ph.D. thesis, U.C. Berkeley, 1997.

[103] T. Palpanas, M. Vlachos, E. J. Keogh, D. Gunopulos, and W. Truppel. Online amnesic approximation of streaming time series. In *Proceedings of the 20th International Conference on Data Engineering(ICDE)*, pages 338–349, March-April 2004.

[104] T. Palpanas, M. Vlachos, E. J. Keogh, D. Gunopulos, and W. Truppel. Online amnesic approximation of streaming time series. In *Proc. of ICDE*, 2004.

[105] P. Phaal, S. Panchen, and N. McKee. Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks. *RFC 3176*, September 2001.

[106] J. Quittek, T. Zseby, B. Claise, and S. Zander. Requirements for ip flow information export (ipfix). *RFC 3917*, October 2004.

[107] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[108] M. A. Shah, S. Madden, M. J. Franklin, and J. M. Hellerstein. Java support for data-intensive systems: Experiences building the telegraph dataflow system. *SIGMOD Record*, 30(4):103–114, 2001.

[109] J. Spiegel and N. Polyzotis. Graph-based synopses for relational selectivity estimation. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 205–216, 2006.

[110] M. Sullivan and A. Heybey. Tribeca: A system for managing large databases of network traffic. In *USENIX Annual Technical Conference*, 1998.

[111] N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *Proc. of VLDB*, 2003.

[112] A. Tenenbaum. *Modern Operating Systems, 2nd Edition*. Prentice Hall, 2001.

[113] Traderbot. `http://www.traderbot.com/`.

[114] P. A. Tucker, D. Maier, and T. Sheard. Applying punctuation schemes to queries over continuous data streams. *IEEE Data Eng. Bull.*, 26(1):33–40, 2003.

[115] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.

[116] M. Wang, N. H. Chan, S. Papadimitriou, and T. M. Madhyastha C. Faloutsos. Data mining meets performance evaluation: Fast algorithms for modeling bursty traffic. In *Proc. of ICDE*, 2002.

[117] W. Wang, H. Jiang, H. Lu, and J. Xu Yu. Containment join size estimation: Models and methods. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 145–156, 2003.

[118] M. Weis and F. Naumann. Dogmatix tracks down duplicates in xml. In *Proc. of SIGMOD*, June 2005.

[119] K. Whang, B. T. V. Zenden, and H. M.Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Trans. Database Syst.*, 15(2):208–229, 1990.

[120] David P. Woodruff. Optimal space lower bounds for all frequency moments. In *SODA*, pages 167–175, 2004.

[121] Y. Zhu and D. Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, pages 358–369, August 2002.